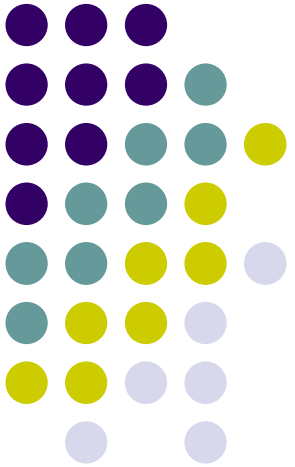
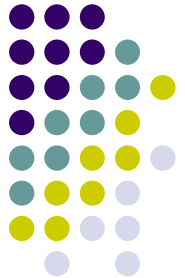
Intensity model with blur effect on GPUs applied to large-scale star simulators

Chao Li

Institute of Software, Chinese Academy of Sciences

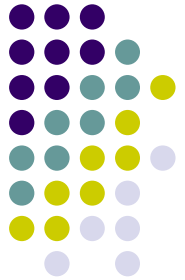


Outline



- Background and Motivation
- Model description
- Our simulators
- Evaluation
- Conclusion
- Future work

Background



- What is star simulator: an important aerospace application, providing star image.
- Star image simulation

  a) position determination

  b) satellite attitude calculation

  c) Navigation feedback

  d) satellite tracker

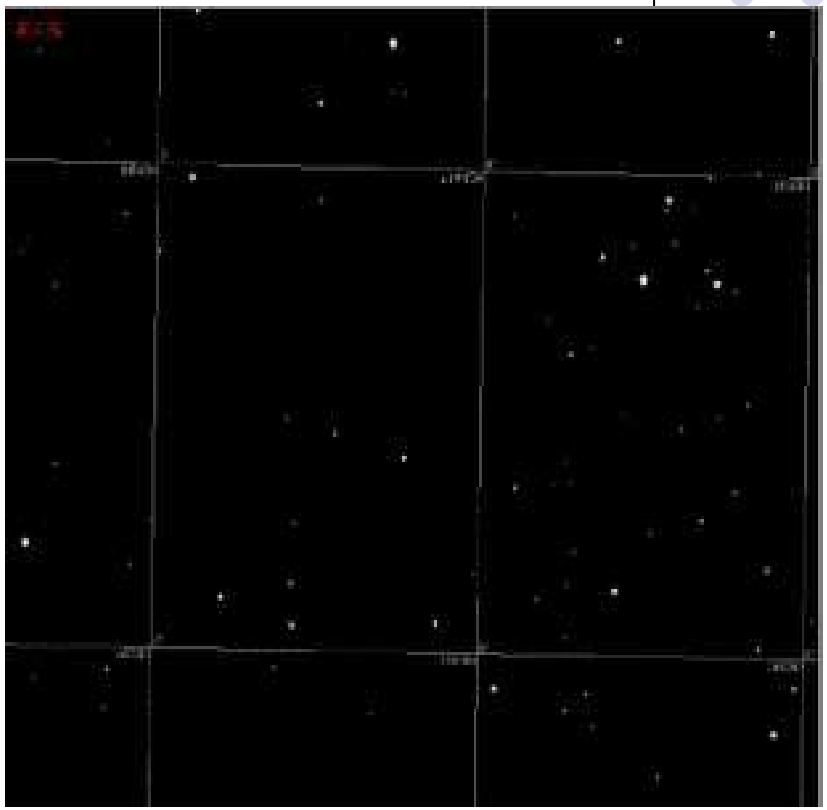
- Intensity model
- Blur effect: Point Spread Function (PSF)
- Problem:

  far from real-time(30 frame/s)

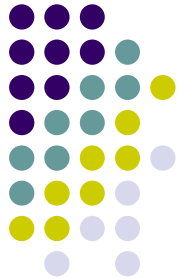
  a) massive algebraic computation

  b). proportional to the number of stars

  c). Computation intensive.

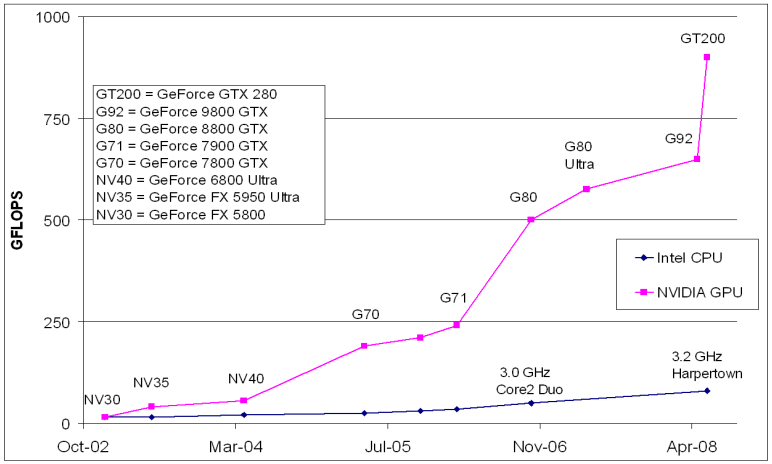


Liebe. C.C (2002), Hye-Young KIM 2002,
Yang Yan-de(2009), Shaodi Zhang(2010)



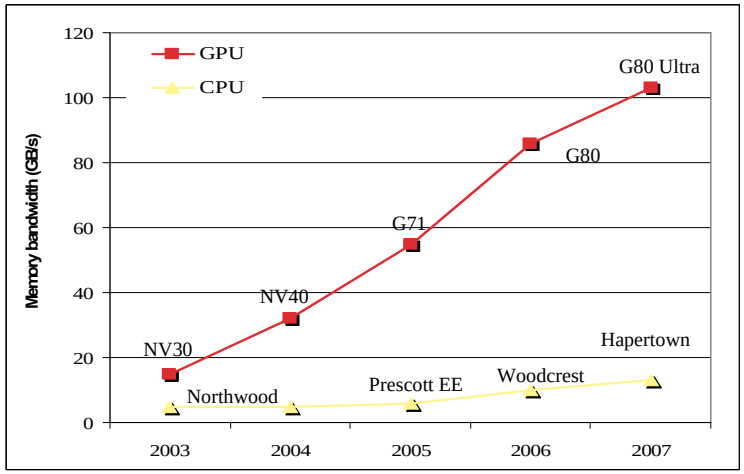
- Pascal
- C
- Prolog
- Sequential system implementation
- Parallelism

GPU Parallel platform



•High Float Computation ability

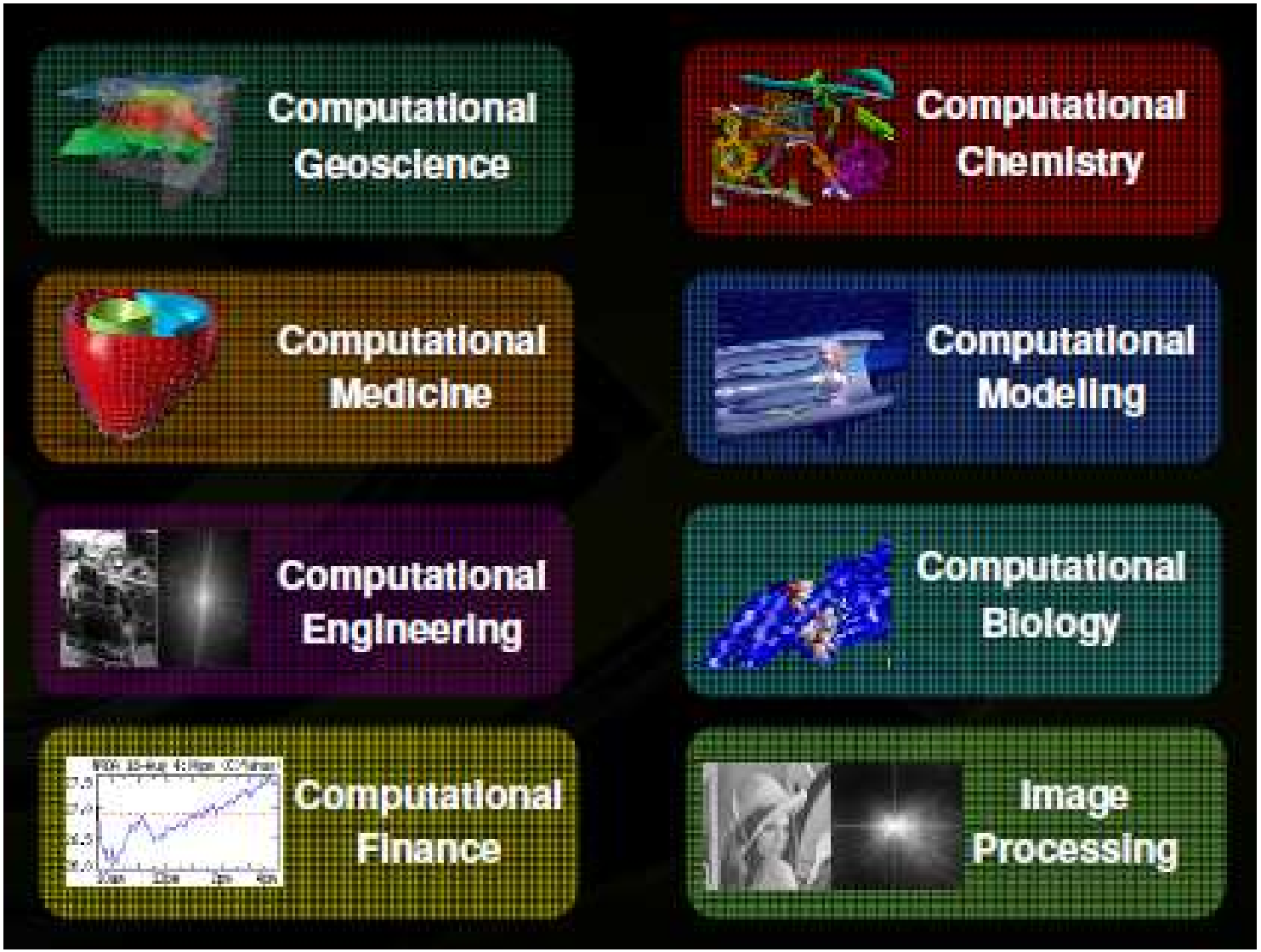
Massive thread running(~10K threads)
Little context switching



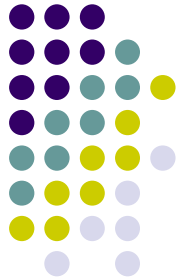
•High Memory Bandwidth

Little memory latency

GPU computing



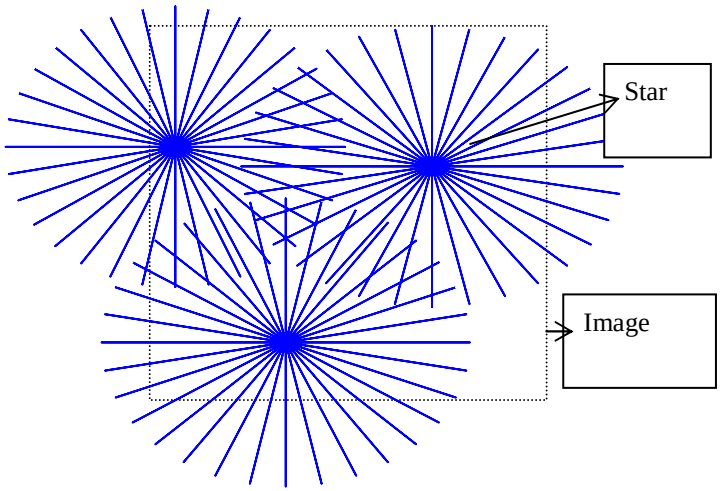
Our work

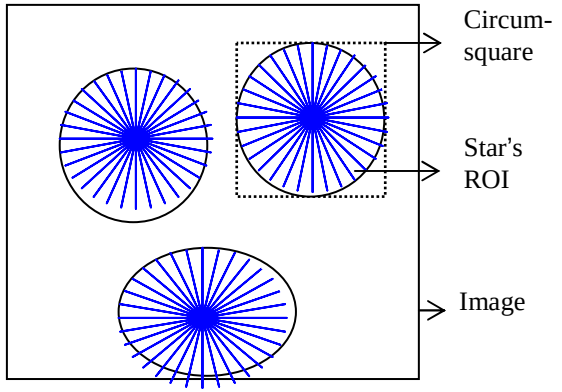


- Parallelization of intensity simulation on GPUs and a parallel star simulator
- Implement a adaptive simulator by adapting parallel simulator to defined problem characteristic with on-chip memory redesign;
- Strategies in achieving high performance of our simulators
- A performance balance analysis to direct the choice of two GPU simulators

What is the model

I





- illustrate the intensity distribution of each star projecting on the space imaging device

- The brightness of a star can be denoted by its magnitude.
- The brightness of a star and its magnitude can be concluded:

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

- Gauss point spread function (Gauss blur effect):

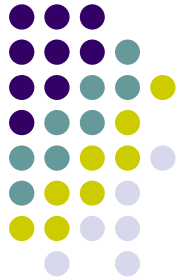
$$\mu(x, y) = \frac{1}{2\pi\delta^2} \exp[-\frac{(x - X)^2 + (y - Y)^2}{2\delta^2}]$$

- Region of Interest (ROI) : a pixel circle centered by star point
- The intensity distribution of a star on a pixel:

$$\varphi(m,x,y) = g(m) \times \mu(x,y)$$

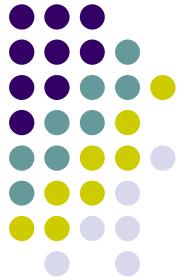
Fig. 2 shows a segment of simulated star image (1024*1024) with 2252 stars projected.

What we have done



- Sequential simulator
- Parallel simulator
- Adaptive simulator

Sequential simulator



- *Star generation*
  1) stars in the FOV of image plane are retrieved from star catalogue
  2) each star contains a magnitude within the range of 0~15 and the coordinate in image plane

- *Star brightness computation*,
  1) calculates the star's brightness following the formula previously explained

- *Pixel computation*,
  1) the computation of gray value of each pixel at the image sequentially

- *Output*
  1) sends out the gray value to form a picture

Simulator Input[1] & output

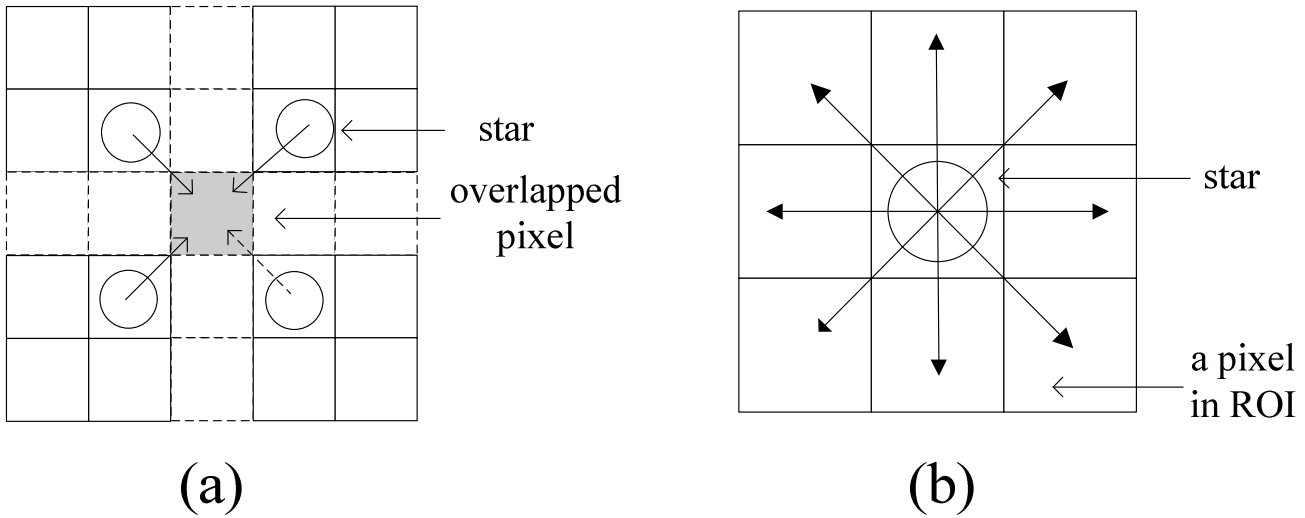
[1]Parallel Accelerating for Star Catalogue Retrieval Algorithm by means of GPUs

Parallel simulator

1) Parallel strategy

- The intensity model computes the gray value of each pixel by accumulating intensity contributions from stars within the ROI
- two alternative approaches to organize parallel execution of the model:

  pixel centric VS star centric

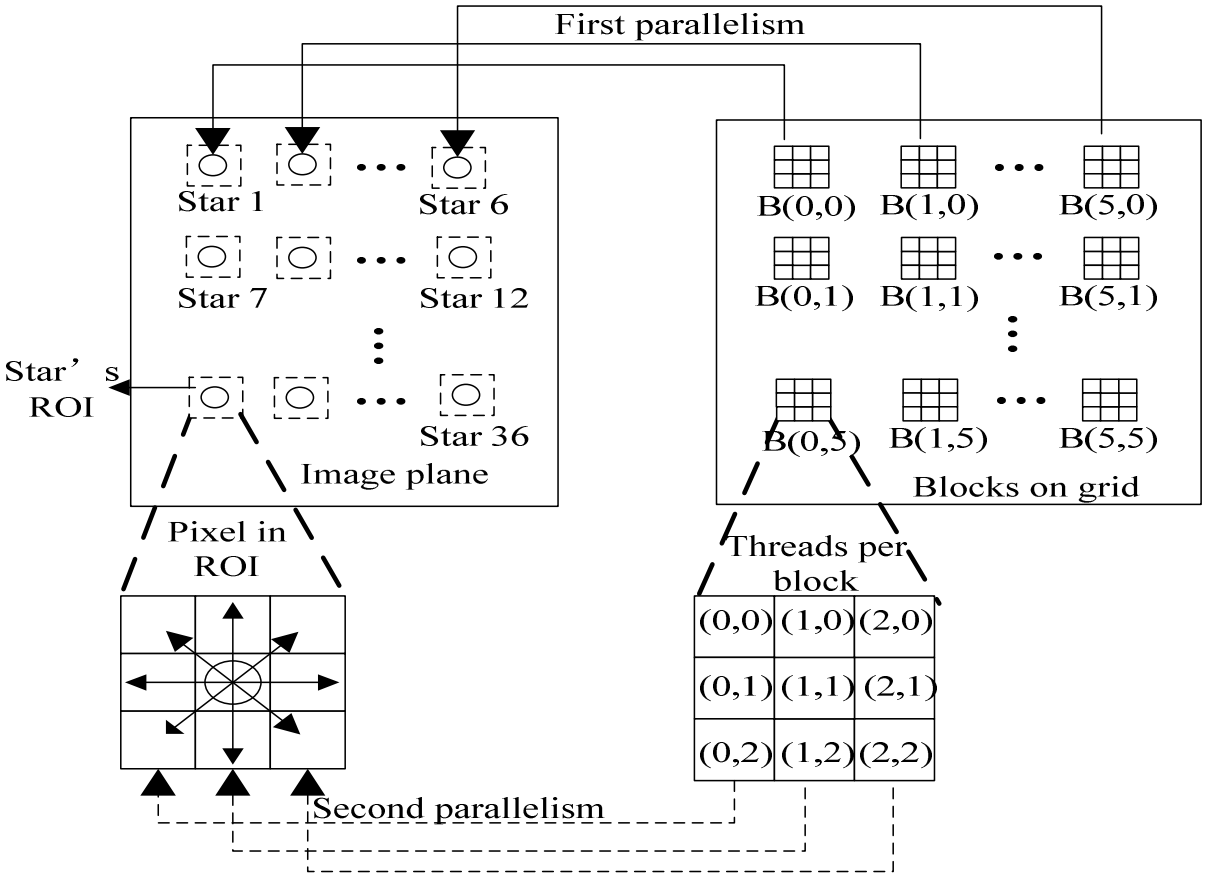


(a) (b)

pixel centric: Generate Thread Divergence
star centric: Eliminate thread divergence;
Need atomic operation

2) Star-centric parallel model

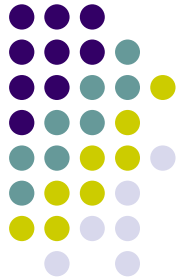
- Different Star distributes in a independent behavior
- The calculation of star's distribution on different pixels is also independent
- Two levels of data parallelism: a) parallelism among stars b) parallelism among pixels inside the ROI of a star



3) GPU implementation

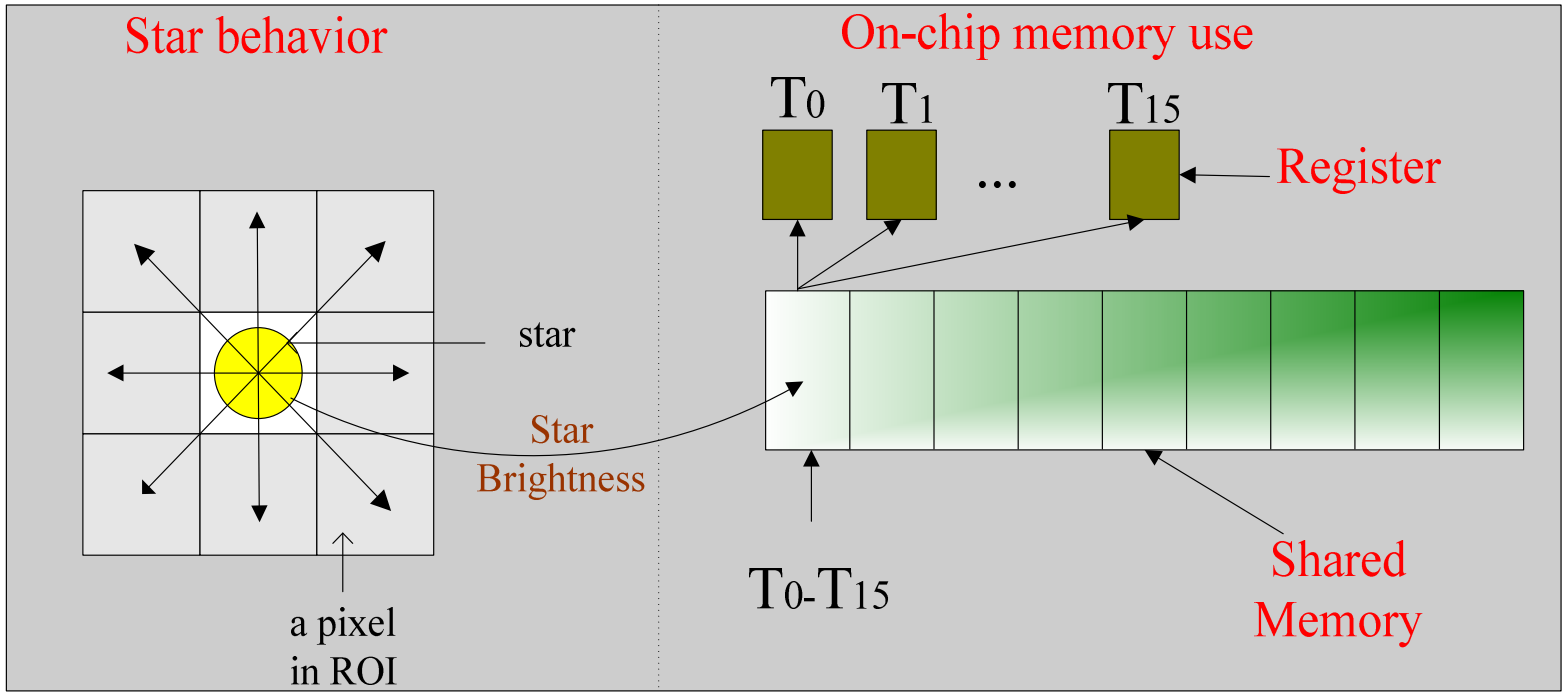
- Star brightness computation; Pixel computation: two computation stages are off-loaded to the GPU to be processed in parallel
- Thread Dimension Determination
- Blocks: 2-dimensional ->support enough stars for simulation
- Threads: 2-dimensional -> simulate the two-loop in star intensity distribution

```
for (pixelY from starPosY-MARGIN to starPosY+MARGIN)        /* determine pixel y-coordinate*/
    for (pixelX from starPosX-MARGIN to starPosX+MARGIN)    /* determine pixel x-coordinate*/
        if (pixelX & pixelY locate in the range of the image)
            starBgt ←calculate star brightness;
            imagePixelArray[pixelY*img_width+pixelX]
                +← calculate pixel gray
```



- Model data : The data containers for stars and pixels
- Indicator elements in the interface of our kernel to prevent the wrong address access of parallel threads

- Data organization on GPU memory:
- For star array: all threads in a warp access the same address.
- For pixel array: spatial locality.

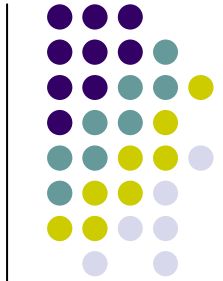


- Shared memory in blocks: each star

  advantage: one shared memory call costs 1~4 clock cycles while a global memory access need 400~600 clock cycles of latency.

- Registers for threads: each pixel

  advantage: relieve the bank collision of share memory generated by different threads accessing it simultaneously

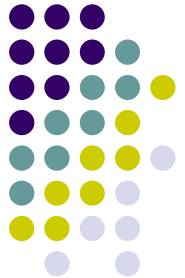


The kernel Pseudo-code of parallel simulator

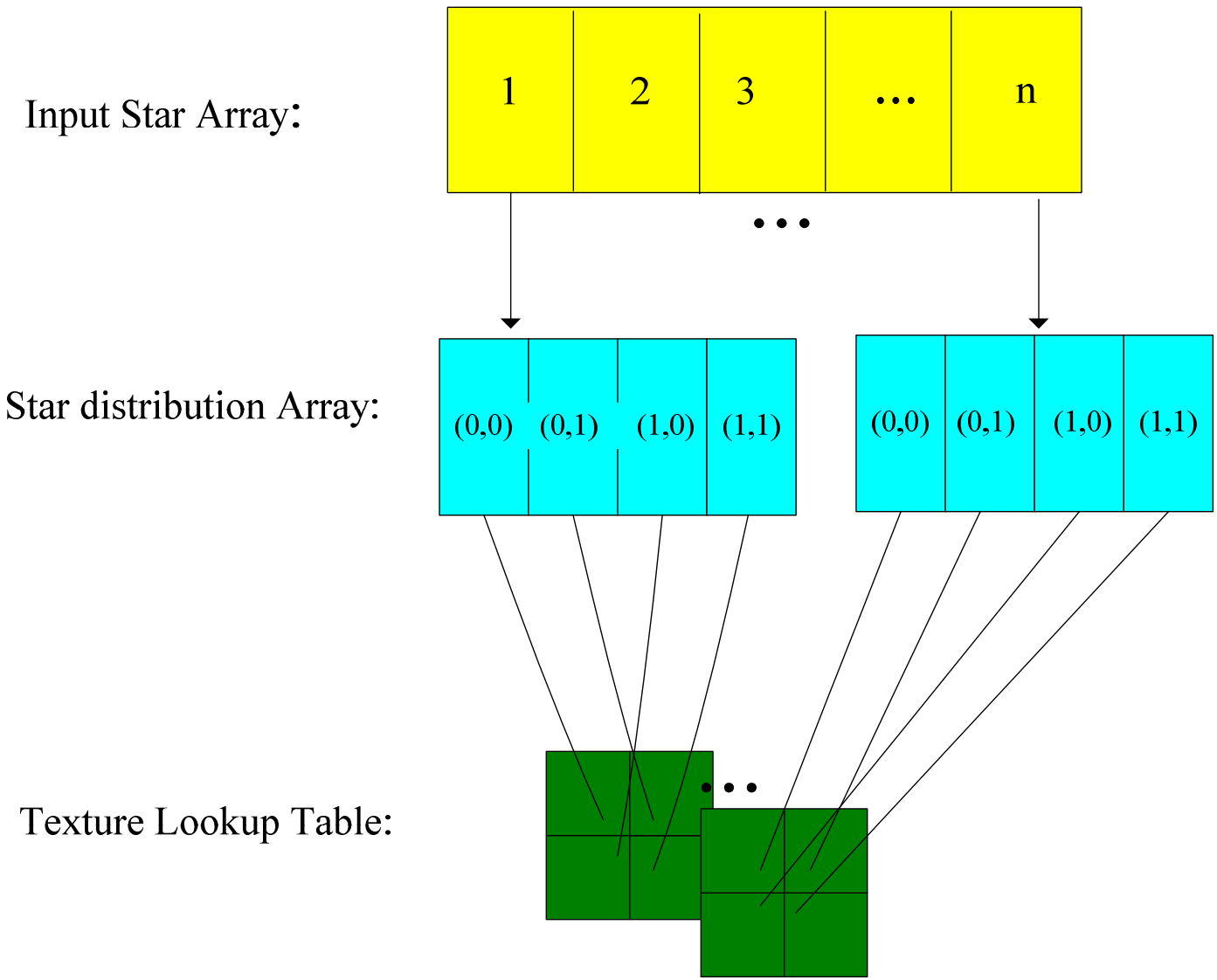
```
Input:   Integer: image_width, image_height, starCount; star*starArray;
Output:  float* imagePixel
1.  __shared__ float  shareMem[3];
2.  threadX ← threadIdx.x, threadY ← threadIdx.y,          /* identify thread index in a thread block*/
    blockId ← blockIdx.x + blockIdx.y*gridDim.x           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

- ROI of different stars within a short distance is likely to overlap
- Atomic add operation
- the stars in simulation are distributed relatively scatter

Adaptive simulator

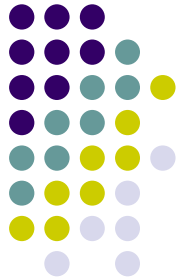


- Fixed star magnitude range: fixed array
- Fixed size of ROI: fixed distribution matrix
- Lookup table: fixed array + fixed distribution matrix
- Build lookup table ahead of kernel, Bound to on-chip texture memory: 1). capitalize 2D spatial locality 2). Cache.
- Shift: computation of distribution to access of lookup table
- Balance between computation and access overhead.



The process of building lookup table.

Evaluation of our simulators

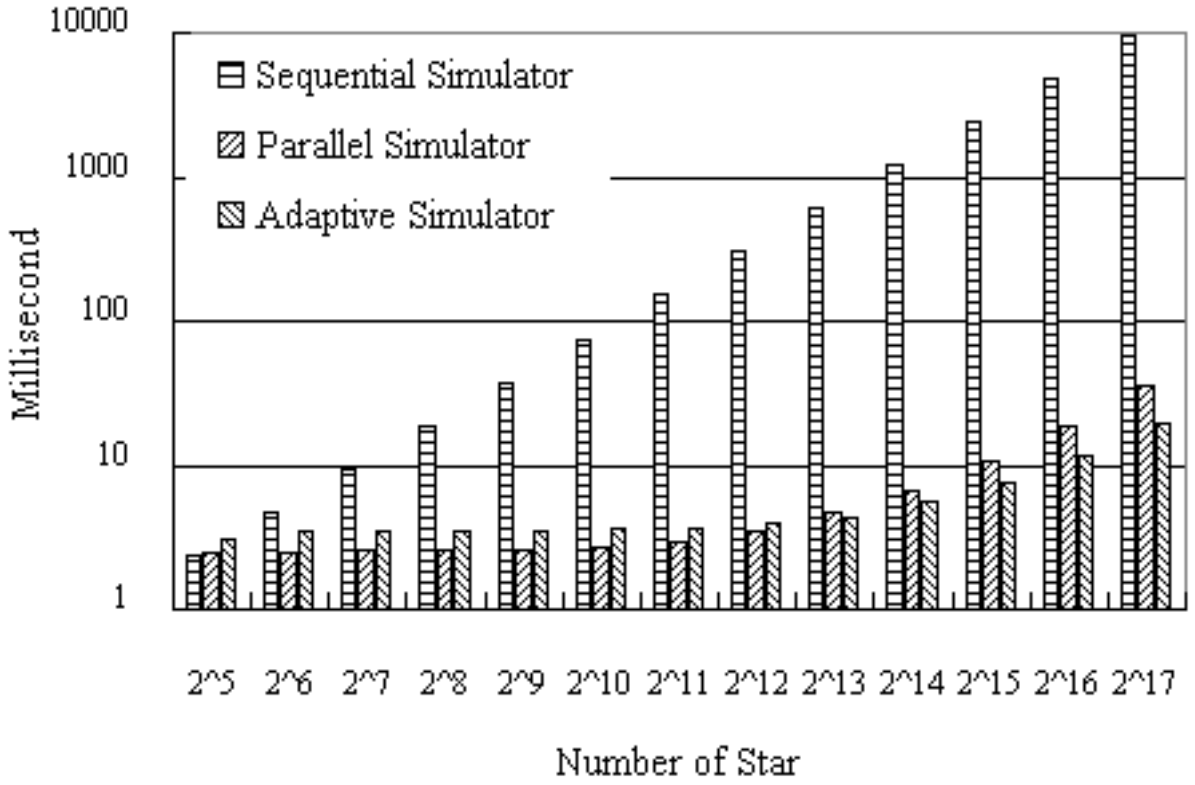


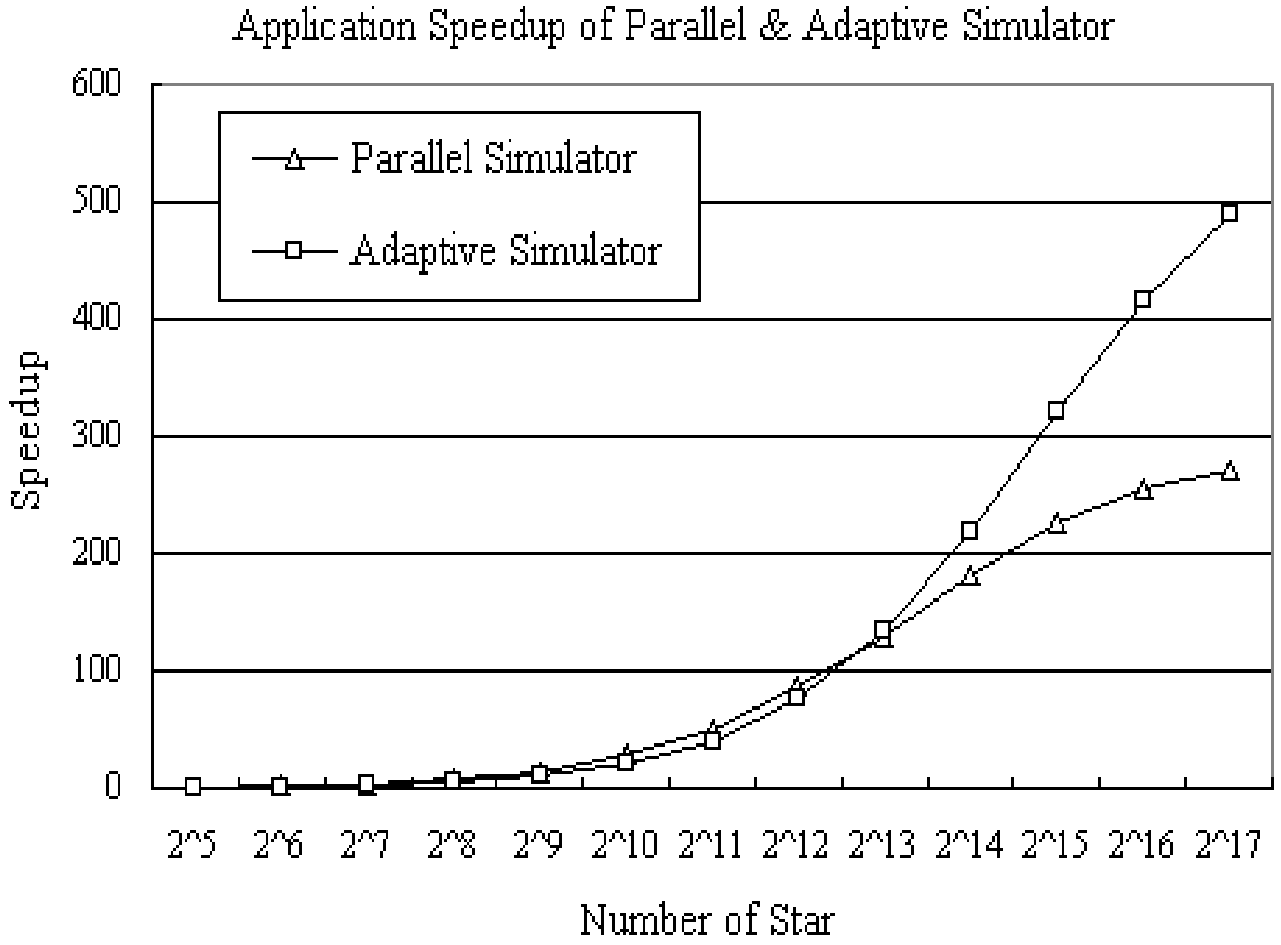
- Benchmark 1
- Benchmark 2
- Selection table
- Discussion

Benchmark 1

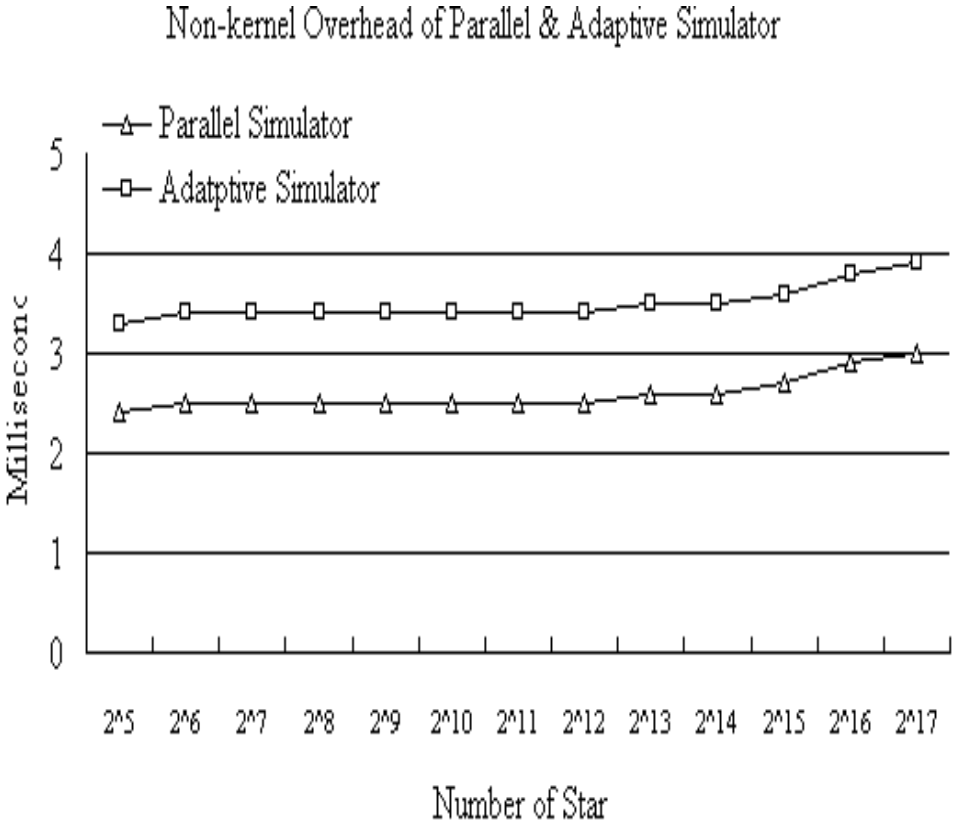
- **Increasing the numbers of stars simulated on the image (and so the number of thread blocks in grid increases)**
- CPU：Intel Core i7 930 2.80GHz，GPU： GeForce GTX480 （FERMI）

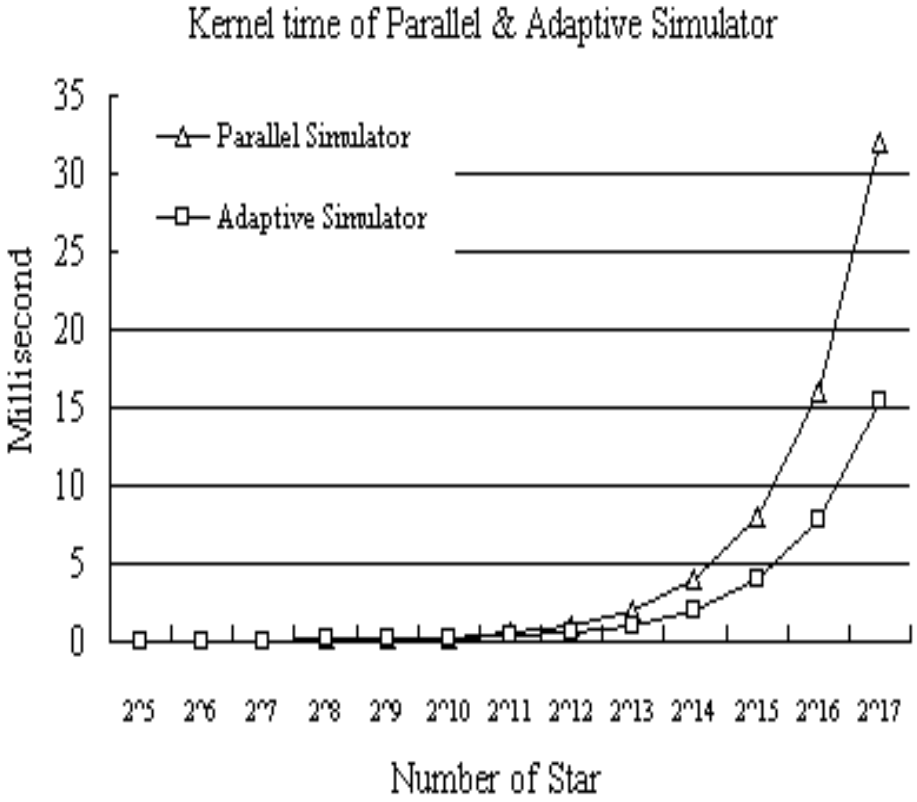
Application performance for sequential, parallel, adaptive simulators: test1





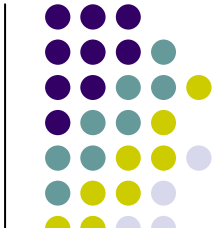
Speedup of parallel simulator, adaptive simulator to sequential simulator: test1





Non-kernel time in parallel & adaptive simulator: test1

Kernel time in parallel & adaptive simulator: test1



The breakdown of non-kernel part for adaptive simulator: test1

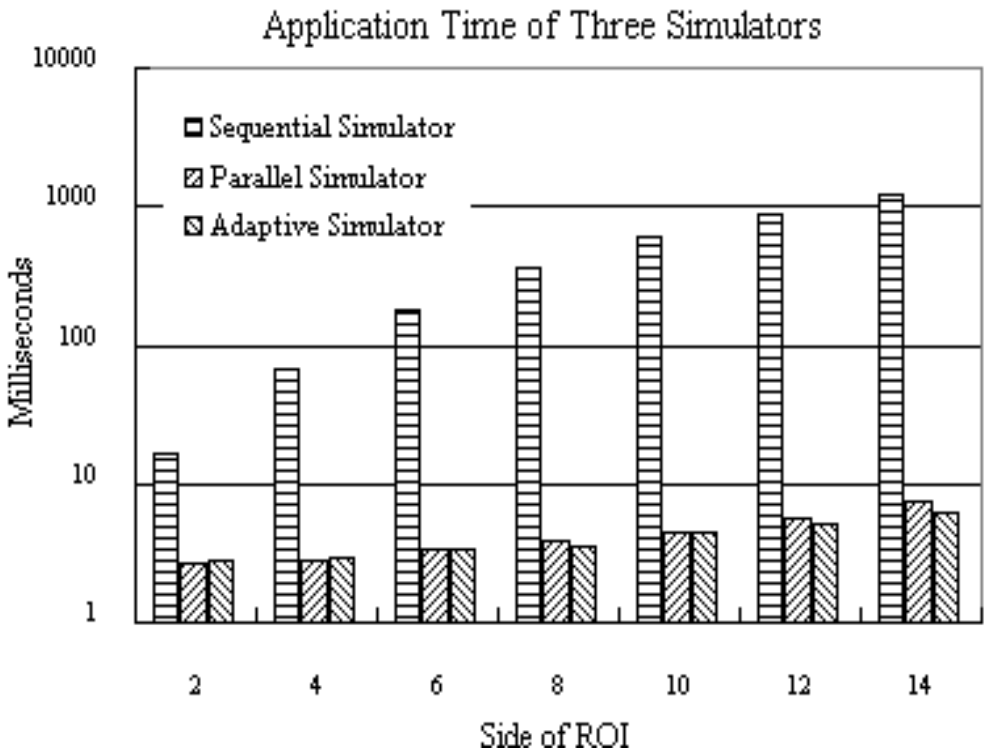
Stars / Time(ms)	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}
CPU-GPU Transmission	2.43	2.45	2.52	2.51	2.50	2.51	2.50	2.51	2.61	2.67	2.71	2.89	3.01
Lookup Table Build	0.70	0.71	0.71	0.72	0.71	0.71	0.70	0.71	0.72	0.71	0.72	0.71	0.71
Texture Memory Binding	0.21	0.20	0.21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

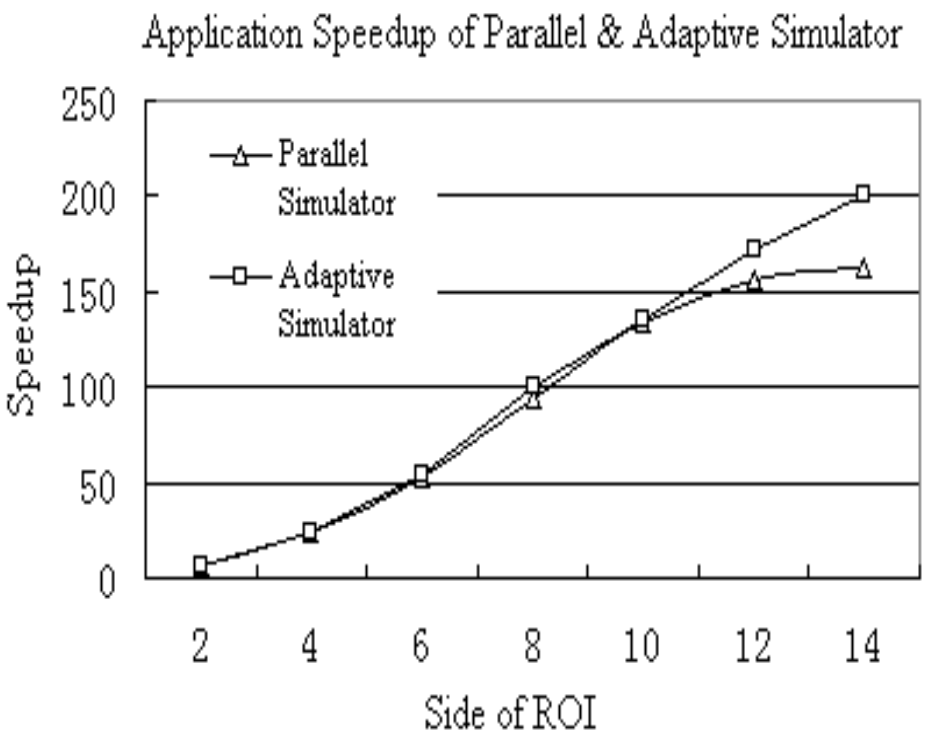
The execution GFlops : test1

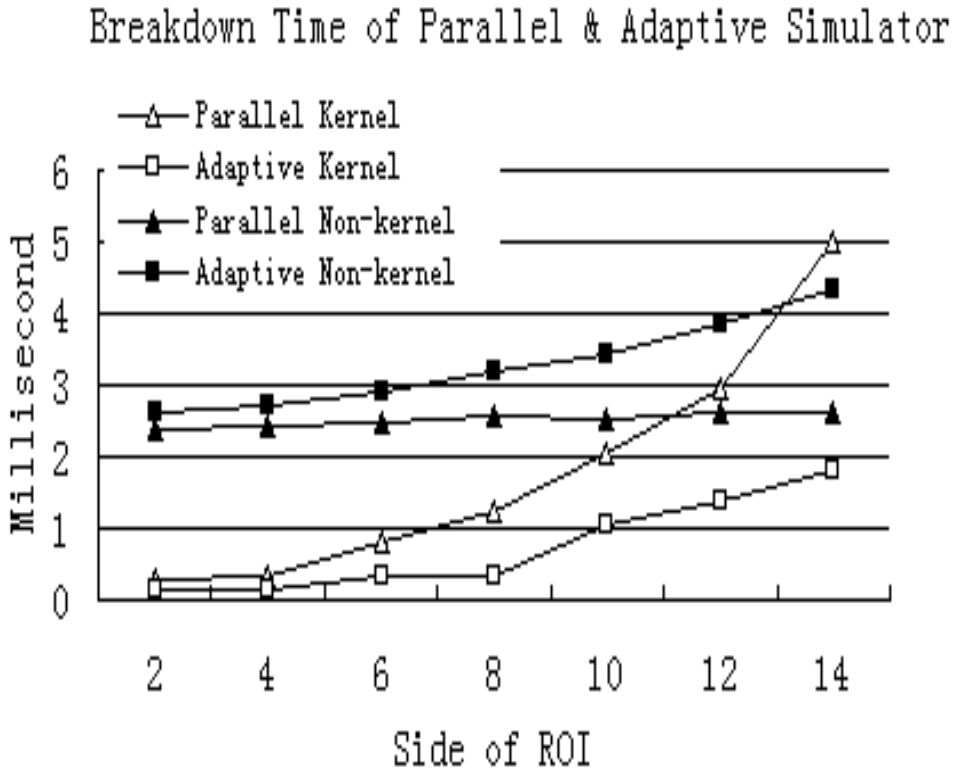
Number of Star	Parallel Simulator (GFLOPS)	Adaptive Simulator (GFLOPS)
2^17	95.07	93.8

Benchmark 2

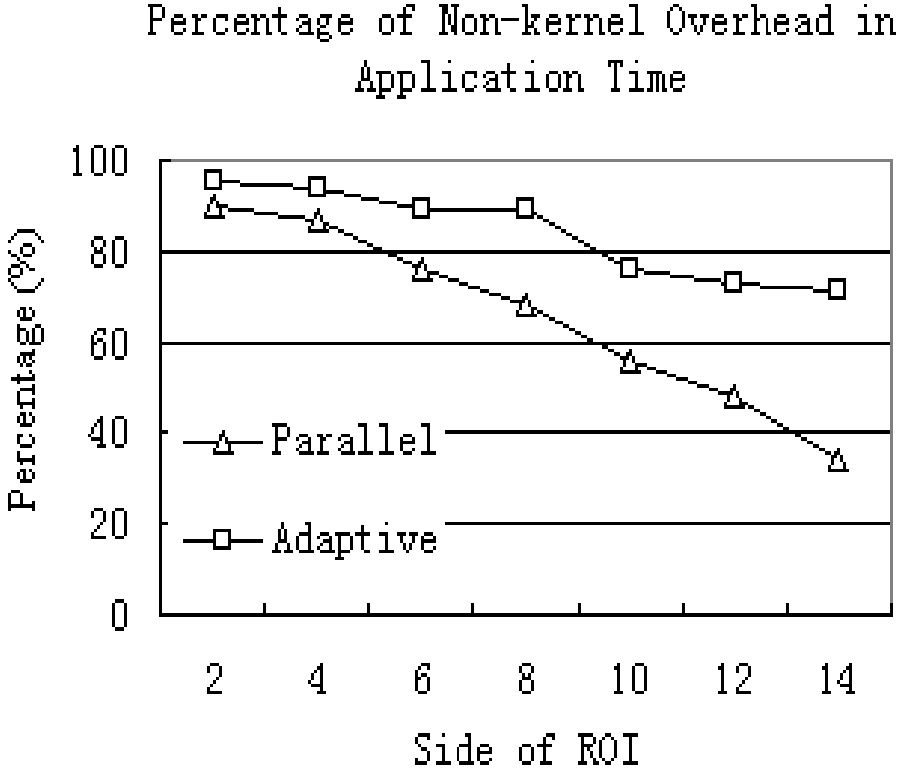
- Increasing the side length of ROI (and so number of threads per thread blocks increases)





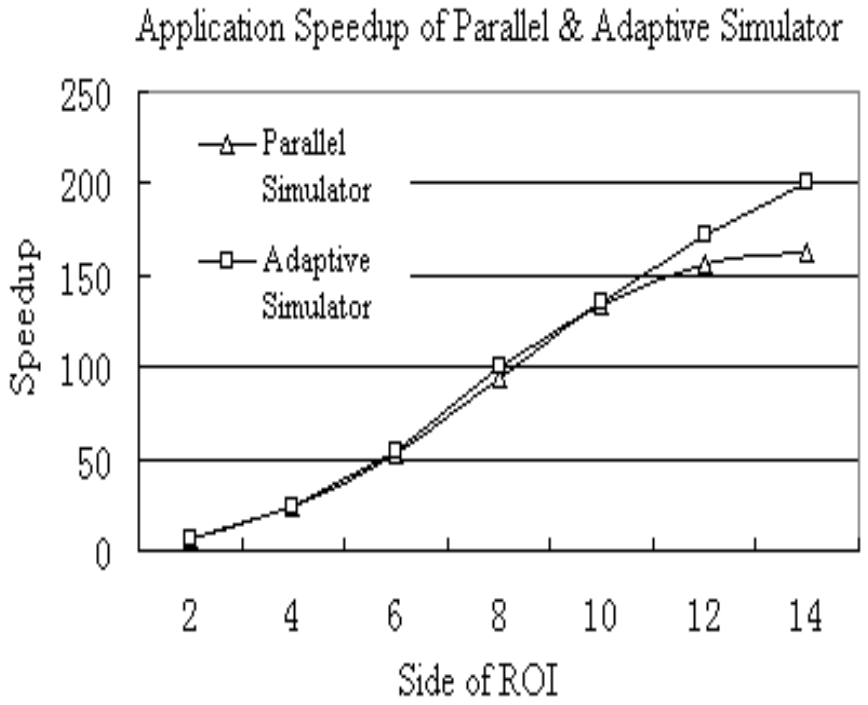


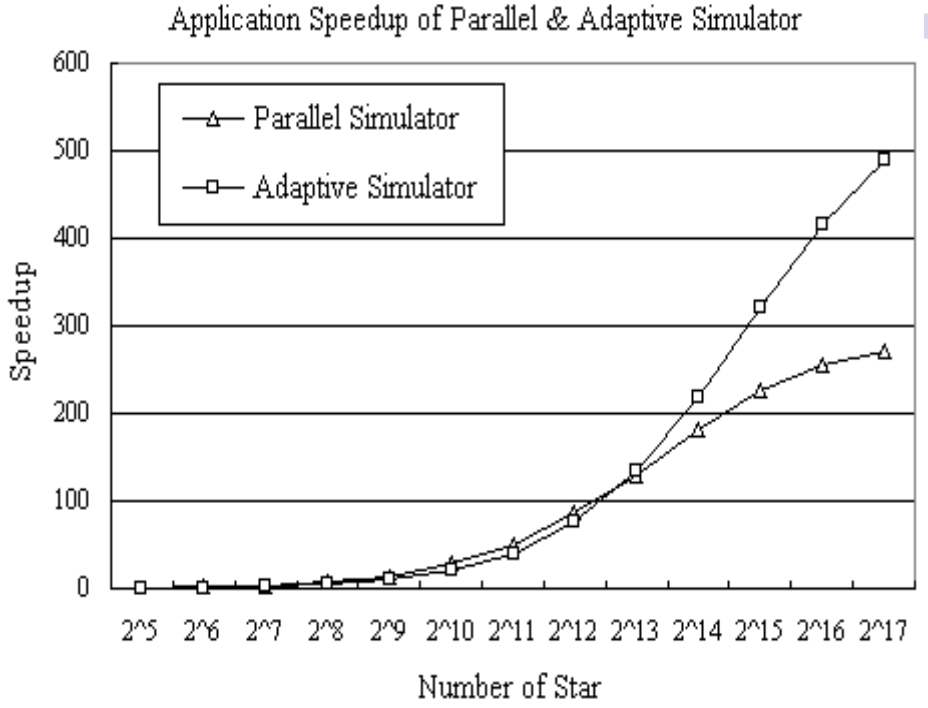
Breakdown of parallel simulator, adaptive simulator: test2



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

Selection table

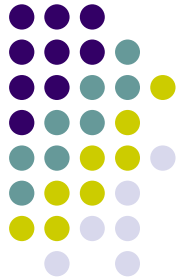




Balance :
the execution of star distribution with fixed star magnitude range from kernel VS
texture memory access by creating a lookup table in texture memory

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

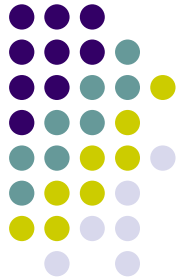
Discussion



- Thread Per Block Restriction on ROI
- Texture Storage Memory Restriction
- Advice on simulators:

  when the star image is in a very small-scale (num of stars : $0\sim2^7$) , the sequential simulator is good.

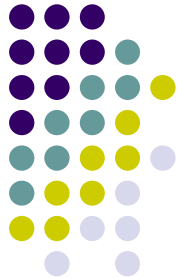
conclusion



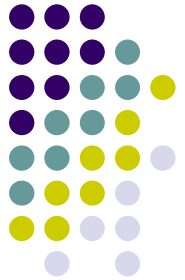
- Three simulators: sequential, parallel, adaptive simulator
- Parallel VS sequential: 270X; Adaptive VS parallel: 1.8X
- results: 1) GPU are good platforms to simulate star image due to the highly data parallelism

  2) parallel simulation behaviors are redesigned by using on-chip textured memory , the performance can be improved
- a balance between the non-kernel overhead and kernel execution; we observe the reflection point and a choice table is given to direct the selection of two simulators.

Future work



- 1. Integrated our work into CSTK
- 2. Scaling our simulator to multi-GPUs.



Thanks!

Any Questions?