DV2575: Projects

Yong Yao

yong.yao@bth.se

DIDD, BTH

General

- Introduction
  - CUDA programming
- Project 1
  - Odd-even sort
- Project 2
  - Gaussian elimination
- Project 3
  - Ray tracing

CUDA programming

- CUDA
  - Compute Unified Device Architecture
  - Freely distributed by NVIDIA
- Basic idea
  - The GPU is linked to the CPU by a reasonably fast connection
  - Use the GPU as a co-processor
- Further
  - Farm out big parallel tasks to the GPU
  - Keep the CPU busy with the control of the execution and “corner” tasks

CUDA programming

- CUDA installation
  - Support various Operating Systems (Oss)
  - I.e., Windows, Mac OSX, Linux
- Quick start guide
  - See document "CUDA_Quick_Start_Guide_v7.5.pdf" on the course page in its-learning
- More questions
  - Feel free to ask Yong Yao (in person) in Lab or office H453A
  - Note that, will not answer by e-mail

CUDA programming

- CUDA programming: extended C

- Declaration specifications:
  - global, device, shared, local, constant
- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads
- Runtime API
  - For memory and execution management
- Kernel launch

```
__device__ float filter[N];
__global__ void convolve (float
*image)  {
  __shared__ float region[M];
  ...
  region[threadIdx.x] = image[i];
  __syncthreads()
  ...
  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per
block
convolve<<<100, 10>>> (myimage);
```

CUDA programming

- Example

```
__global__ void testKernel (void) {
}

int main(void) {
  testKernel<<<1,1>>>();
  return 0;
}
```

- __global__
  - CUDA C/C++ keyword
  - The function runs on the device, and is called from host code
- main()
  - Processed by standard host compiler, e.g., gcc
- testKernel()
  - Processed by NVIDIA compiler

CUDA programming

- testKernel<<<1,1>>>();
  - Triple angle brackets mark a call from *host* code to *device* code
  - This is required to execute a function on the GPU
- <<<1,1>>>
  - The host instructs an execution configuration
  - At GPU, it is supposed to run 1block, each with 1 thread
  - A block represents the entity that gets executed by an SM (stream multiprocessor)
  - Each block cannot has larger than 1024 threads
- <<<x,y,z>>>
  - 3D structure

CUDA programming

- CUDA function declarations
  - __global__: must return void
  - Others, see CUDA reference manual

	Executed on the	Only callable from
__device__ float myDeviceFunc()	device	device
__global__ void myKernelFunc()	device	host
__host__ float myHostFunc()	host	host

CUDA programming

- Important APIs
- cudaMalloc()
  - Allocates object in the device Global Memory
  - Para1: address of a pointer to the allocated object
  - Para2: size of allocated object
- cudaFree()
  - Frees object from device Global Memory
  - Para1: pointer to freed object
- cudaMemcpy()
  - Frees object from device Global Memory
  - Pointer to freed object

CUDA programming

- Important APIs (cont.)
- cudaMemcpy()
  - Memory data transfer
  - Para 1: pointer to source
  - Para 2: pointer to destination
  - Para 3: number of bytes copied
  - Para 4: type of transfer
    - Host to Host
    - Host to Device
    - Device to Host
    - Device to Device

CUDA programming

- Matrix manipulation

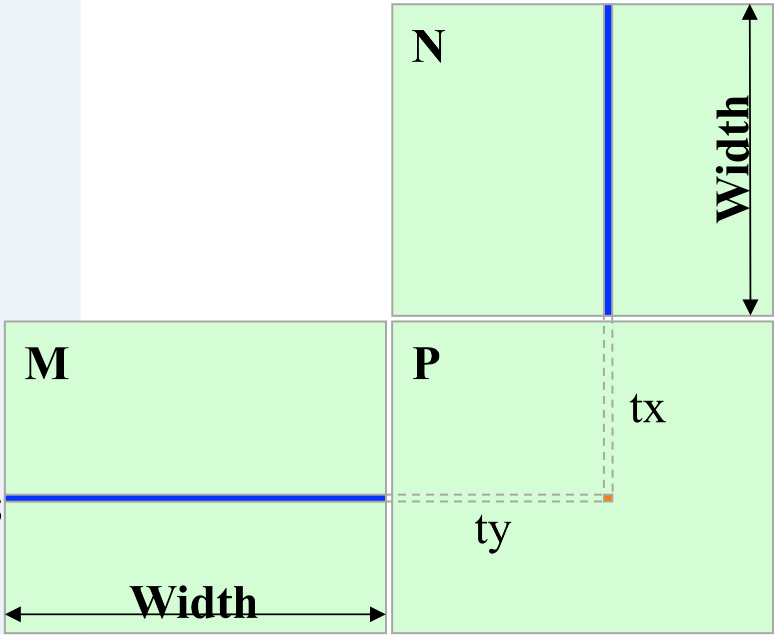
```
__global__ void MatrixMulKernel(Matrix M, Matrix N,
Matrix P) {
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  float Pvalue = 0;

  for (int k = 0; k < M.width; ++k)  {
    float Melement = M.elements[ty * M.width + k];
    float Nelement = N.elements[k * N. width + tx];
    Pvalue += Melement * Nelement;
  }

  P.elements[ty * P. width + tx] = Pvalue;
}

void main() {
  …
  MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);
  …
}
```



Project 1: odd-even sort

- General concept
  - An odd–even sort is a comparison sort related to bubble sort.
  - It functions by comparing all odd/even-indexed pairs of adjacent elements in the unsorted list of numbers.
- Comparison goal
  - Make each pair of numbers in the expected order by doing switch operation
  - (e.g., the first number is smaller than the second one) n
- Comparison rules
  - Repeating the comparison for each even(/odd)-indexed pairs
  - Alternating the comparison between odd->even and even->odd steps.

Project 1: example

Unsorted

3	2	3	8	5	6	4	1	Phase 1 (odd)
2	3	3	8	5	6	1	4	Phase 2 (even)
2	3	3	5	8	1	6	4	Phase 3 (odd)
2	3	3	5	1	8	4	6	Phase 4 (even)
2	3	3	1	5	4	8	6	Phase 5 (odd)
2	3	1	3	4	5	6	8	Phase 6 (even)
2	1	3	3	4	5	6	8	Phase 7 (odd)
1	2	3	3	4	5	6	8	Phase 8 (even)
1	2	3	3	4	5	6	8	

Sorted

Project 1: odd-even transposition

- The algorithm is guaranteed to terminate after N/2 odd-even and even-odd steps.
- After n phases of odd-even exchanges, the sequence is sorted.
- Each phase of the algorithm (either odd or even) requires $\Theta(n)$ comparisons.
- Serial complexity is $\Theta(n^2)$.

Project 1: parallel transposition

- Consider the one item per processor case.
- There are n iterations, in each iteration, each processor does one compare-exchange.
- The parallel run time of this formulation is $\Theta(n)$.
- This is cost optimal with respect to the base serial algorithm but not the optimal one.

Project 1: tasks

- Implement the single thread based odd-even sort
- Implement the multiple-thread based odd-even sort
- Compare the time used for soring in the two methods
- Performance analysis showing how the following factors affect the performance:
  - Effect of number of integers on sorting.
  - Effect of number of threads on sorting.
- One student per implementation
- Evaluation: G/U

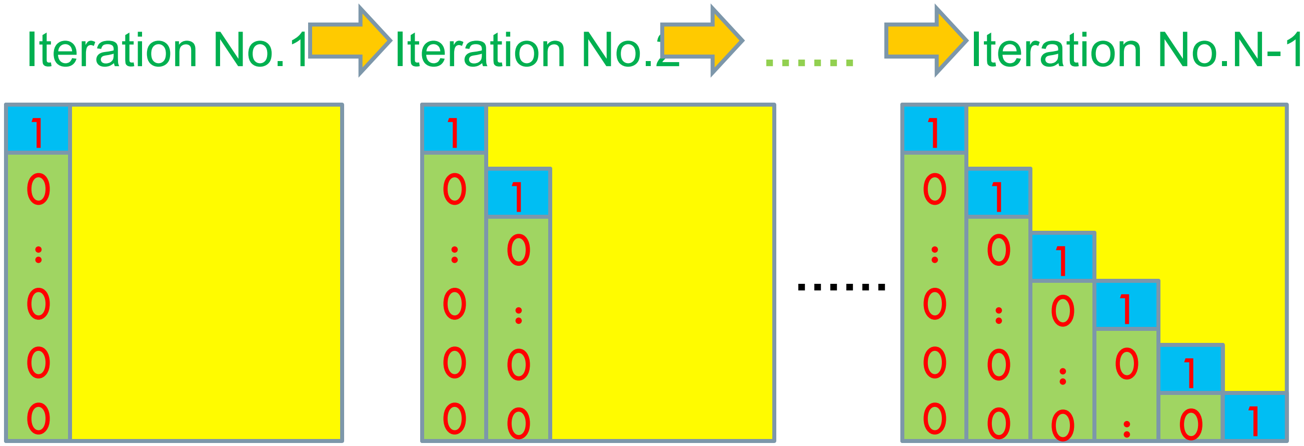
Project 2: gaussian elimination

- General concept
  - An algorithm to solve linear equations.
  - Conduct a sequence of operations on the matrix with coefficients
- Solution approach
  - Transform linear equations into an upper-triangular matrix
  - A pivot column is used to modify the matrix until the lower left-hand corner is filled with zeros, as much as possible.
  - After the transformation, back-substitution is applied.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix},$$

Project 2: gaussian elimination

- Solution approach (cont.)
  - Forward substitution



  - Back substitution, for vector (x_1, x_2,…, x_{k-1}, x_k) then compute the result from x_k to x_1 , i.e., x_k -> x_{k-1} ->…->x_2->x_1

Project 2: example

$$\begin{cases} 3x - 2y + 2z = 16 \\ 7x - 3y + 2z = 26 \\ 2x - y + 4z = 18 \end{cases}$$

Project 2: example

Forward substitution

$$\begin{cases} 3x - 2y + 2z = 16 \\ 7x - 3y + 2z = 26 \\ 2x - y + 4z = 18 \end{cases} \quad *(-2)+$$

Project 2: example

Forward substitution

$$\begin{cases} 3x - 2y + 2z = 16 \\ x + y - 2z = -6 \\ 2x - y + 4z = 18 \end{cases}$$

Project 2: example

Forward substitution

$$\begin{cases} x + y - 2z = -6 \\ 2x - y + 4z = 18 \\ 3x - 2y + 2z = 16 \end{cases}$$

Project 2: example

Forward substitution

$$\begin{cases} x + y - 2z = -6 \\ 2x - y + 4z = 18 \\ 3x - 2y + 2z = 16 \end{cases}$$

$*(-2)+$

$*(-3)+$

Project 2: example

Forward substitution

$$\begin{cases} x + y - 2z = -6 \\ -3y + 8z = 30 \\ -5y + 8z = 34 \end{cases} \quad *\left(-\frac{5}{3}\right)+$$

Project 2: example

Forward substitution

$$\begin{cases} x + y - 2z = -6 \\ -3y + 8z = 30 \\ -\dfrac{16}{3}z = -16 \end{cases}$$

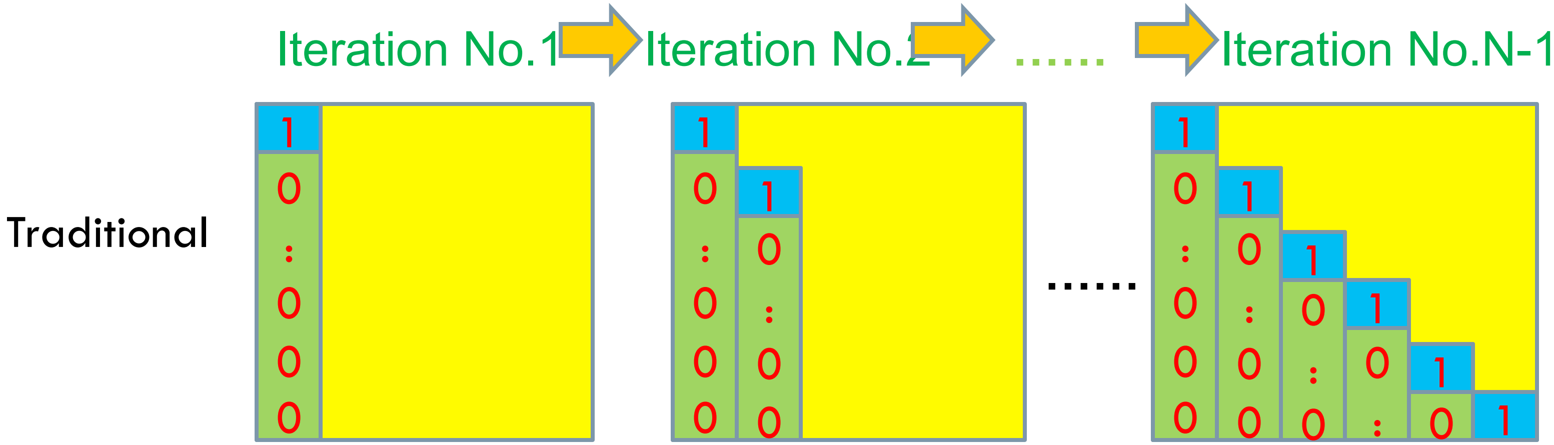
Project 2: example

Back substitution

$$\begin{cases} x + y - 2z = -6 \\ -3y + 8z = 30 \\ -\dfrac{16}{3}z = -16 \end{cases} \Rightarrow \begin{cases} z = 3 \\ y = -2 \\ x = 2 \end{cases}$$

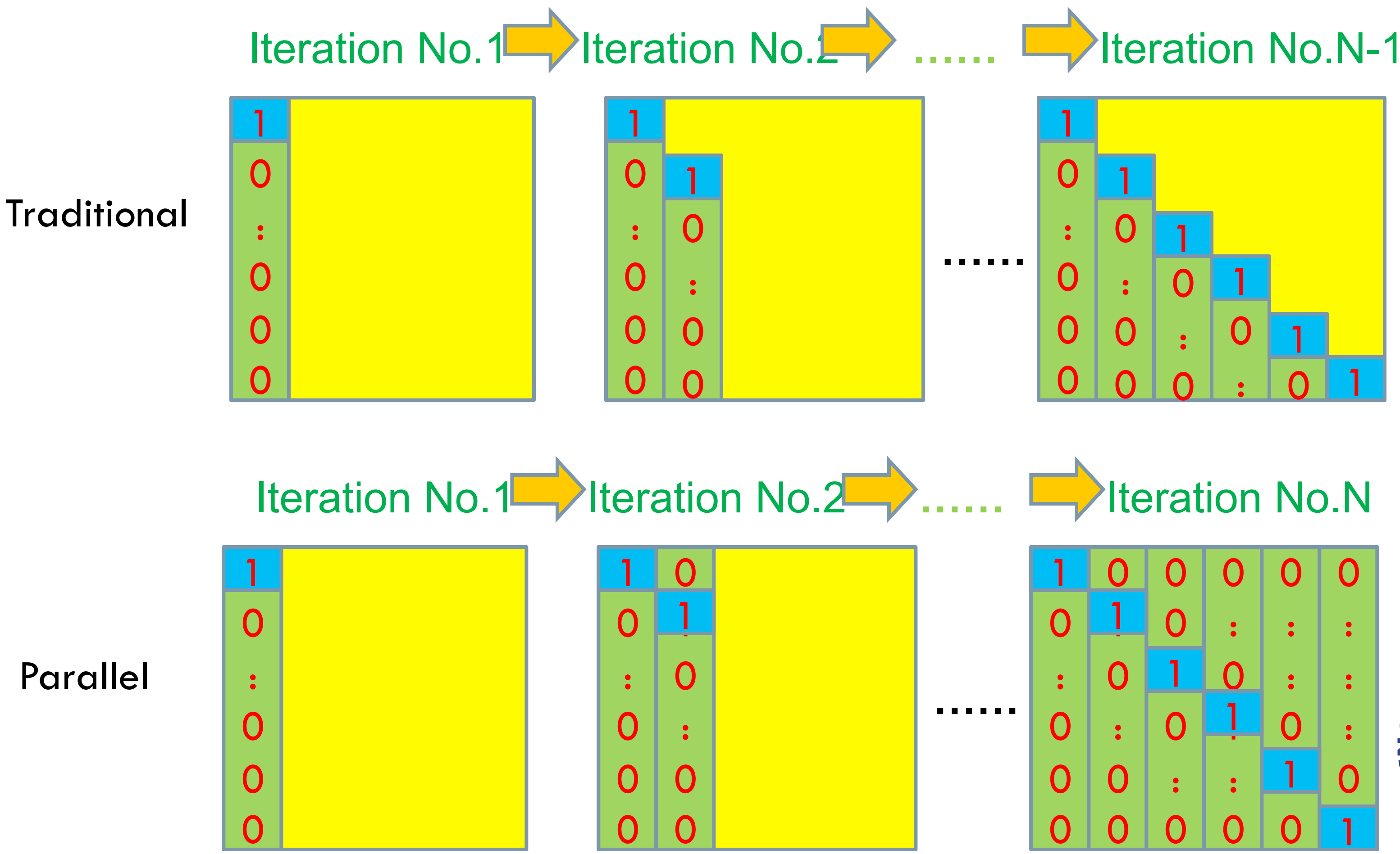
Project 2: CUDA based solution

- Hits: combine forward- and back-substitution



Project 2: CUDA based solution

- Hits: combine forward- and back-substitution



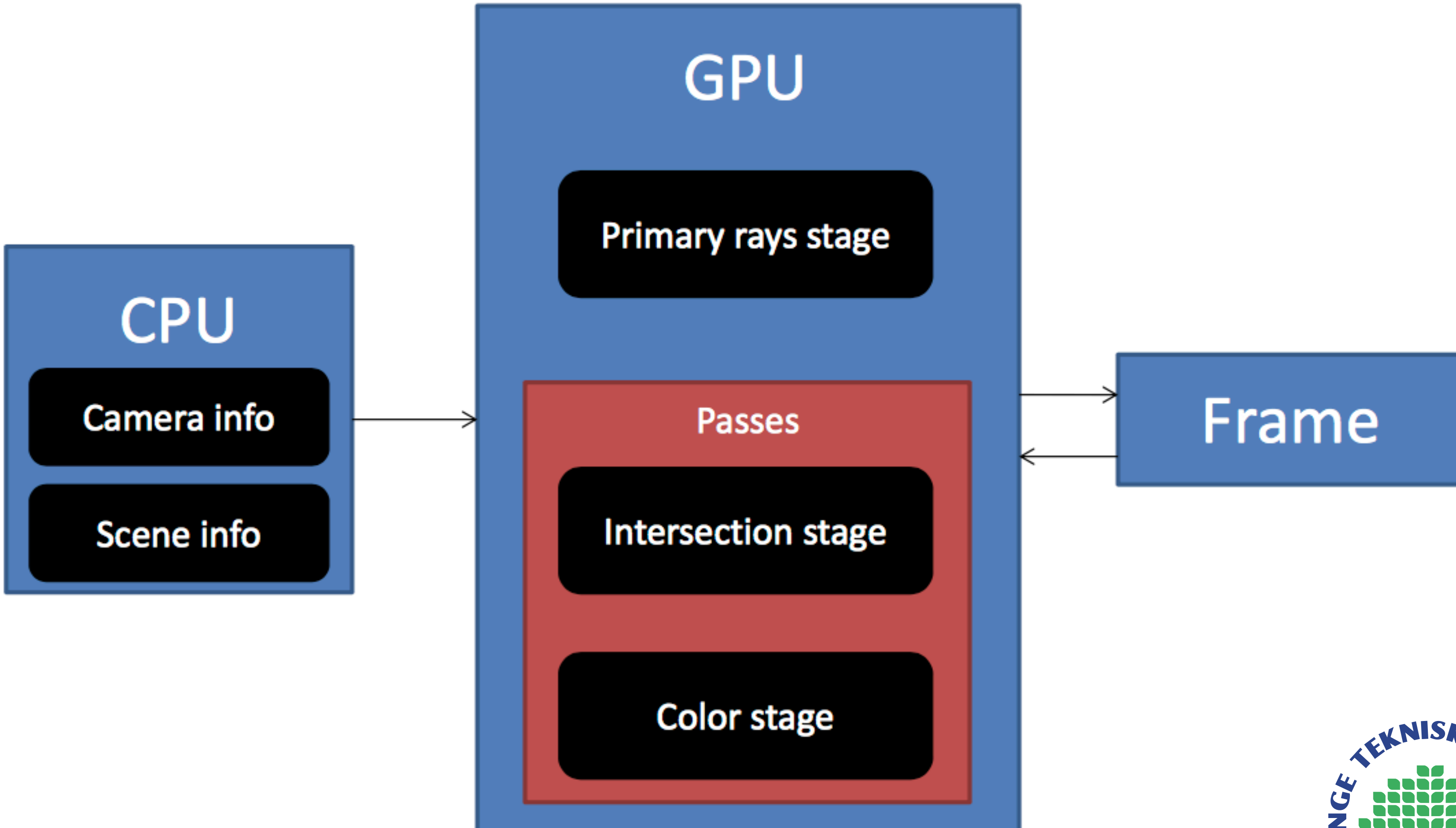
Project 2: tasks

- Implement the single thread based gaussian elimination
- Implement the multiple-thread based gaussian elimination
- Compare the time used in the two methods
- Performance analysis showing how the following factors affect the performance:
  - Effect of number of variables on solving linear equations.
  - Effect of number of threads on solving linear equations.
- One or two students per implementation
- Evaluation: G/U

Project 3: ray tracing

- Ray tracing
  - It is often used to render realistic images where rendering time is less important.
  - By using current generation of GPGPU hardware it is possible to perform ray tracing in real time.
- Inspired by book chapters
  - Interactive Ray Tracing Using the Compute Shader in DirectX 11
  - Bit-Trail Traversal for Stackless LBVH on DirectCompute

Project 3: system



Project 3: tasks

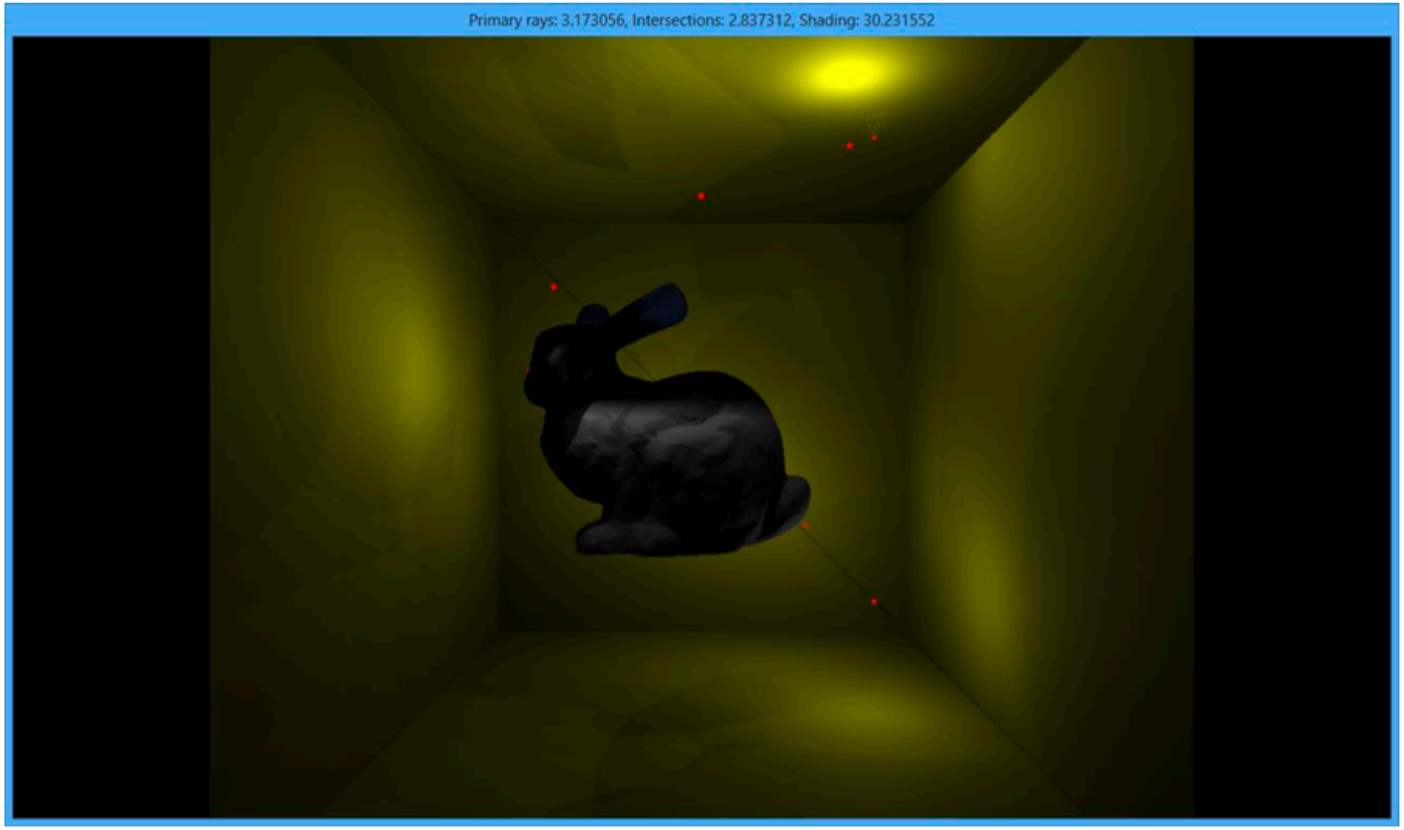
- Generate primary rays from a camera position and orientation.
- Support diffuse lighting with light attenuation Implement the single thread based odd-even sort
- Support multiple ray traces/bounces Implement the multiple-thread based odd-even sort
- Support ray tracing of one triangle mesh

Project 3: tasks (cont.)

- Performance analysis showing how the following factors affect the performance:
  - Number of threads per thread group.
  - Screen resolution.
  - Trace depth.
  - Number of light sources.
- One or two students per implementation
- Evaluation: G/U

Project 3: Example 1

- 10 lights – 1 trace
  A good tip is to generate the box automatically (same buffer)!



Project 3: Example 2

- 10 lights – 10 trace