Multicore Computing

Weijie Zhao

08/29/2024

OpenMP

- Open Multi-Processing
- An API that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran

OpenMP: Quick Start

```
for (int i = 0; i < N;++i){
        b[i] = a[i] + 1;
}
```

OpenMP: Quick Start

```
#pragma omp parallel for schedule(static) num_threads(8)
for (int i = 0; i < N;++i){
        b[i] = a[i] + 1;
}
```

- g++ test.cc –fopenmp -o test -O2

- brew install libomp
- clang++ test.cc -o test -O2 -Xpreprocessor -fopenmp -I/usr/local/include -L/usr/local/lib -lomp

OpenMP: Quick Start

```
int sum = 0;


for (int i = 0; i < N; ++i){
        sum += a[i];
}
```

OpenMP: Quick Start

```
int sum = 0;
#pragma omp parallel for schedule(static) default(shared) reduction(+:sum) num_threads(8)
for (int i = 0; i < N; ++i){
        sum += a[i];
}
```

OpenMP: Slow Start

- #include<omp.h>
- void omp_set_num_threads(int num_threads)
- int omp_get_num_threads()
- int omp_get_thread_num()

- #pragma omp atomic (update/read/write/capture)
- #pragma omp critical

Gauss-Seidel Smoother

- Solving PDE

- Parallel
- Synchronization
- Lock
- Communication

Matrix Multiplication (in Theory)

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

Matrix Multiplication (in Theory)

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

Arithmetic intensity: the ratio of the work to the memory traffic

Strassen Algorithm

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}); \\
M_2 &= (A_{21} + A_{22})B_{11}; \\
M_3 &= A_{11}(B_{12} - B_{22}); \\
M_4 &= A_{22}(B_{21} - B_{11}); \\
M_5 &= (A_{11} + A_{12})B_{22}; \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}); \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}),
\end{aligned}
$$

$$
\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}
$$

HW 1: Logistic Regression

- Given matrix X and label Y, perform gradient descent of logistic regression
- 10 independent test cases. Each case weights 1 pt.

- The compilation is considered failed if it does not finish in 1 minute.
- A test case is considered incorrect if it does not finish in 2 minutes.
- The training accuracy must reach 60%.

- The summation of the execution time across 10 cases will be uses to rank correct solutions.

- Due: 09/13/2024 5:00 pm EDT

Grading

- Homework 40%
- Reading 10%
- Project 50%

- 90%<=A<=100%
- 80%<=B<90%
- 70%<=C<80%
- 60%<=D<70%
- 0%<=F<60%

- 5 pieces of homework.
- No late submissions.
- No 3rd party code
- Automatically tested: Please strictly follow the output format. An incorrect format is considered as a wrong answer.
- The best 4 scores among the 5 are counted in your final grade.
- The fastest correct solution in each homework gets 10% bonus score in the final grade.
- Other correct solutions that are no slower than 2X of the fastest one gets 5% bonus score in the final grade.

Disclaimer: the instructor reserves the right to end or modify any grading criteria at any time.

Input Data

- First line contains 8 integers: N D x0 x1 A B C M
- For i >= 2
  - X[i] = (A * X[i - 1] + B * X[i - 2]+ C) % M
- For all i
  - X[i] /= M;

- N <= 10^5
- D <= 1600

Input Data

- First line contains 8 integers: N D x0 x1 A B C M
- For i >= 2
  - X[i] = (A * X[i - 1] + B * X[i - 2]+ C) % M
- For all i
  - X[i] /= M;

Caution the potential overflow here!

- N <= 10^5
- D <= 1600

Output Format

- D lines
- Each line contains a floating number
  - The logistic regression parameters

What Do We Need to Do?

- We are required to complete two scripts
- compiler.sh
  - it is executed once before the actual testing starts
- run.sh
  - it should takes two arguments, the first argument is the input file name, the second one is the file name that you should write your results into.

Testing Environment

- ssh yourusername@granger.cs.rit.edu
- Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
- 48 threads in total (2 sockets, 12 cores per socket, 2 threads per core)
- 251 GB memory

- Testing limit:
  - 8 threads   taskset -c

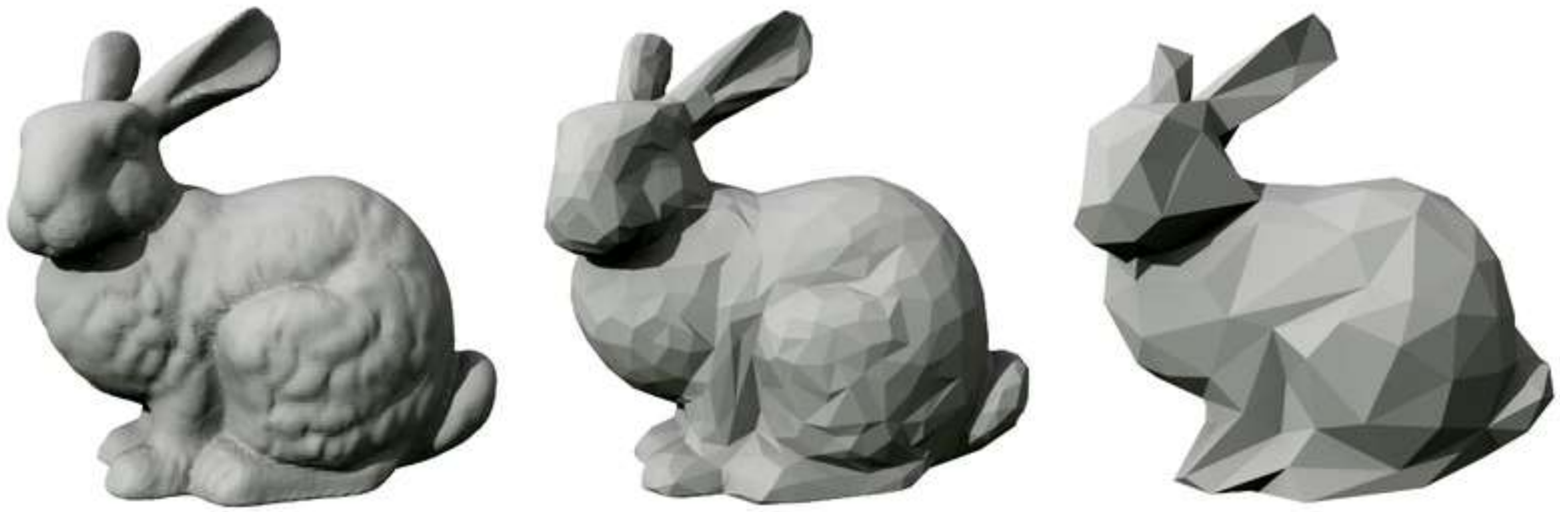
“Premature optimization is the root of all evil”

--- Sir Tony Hoare

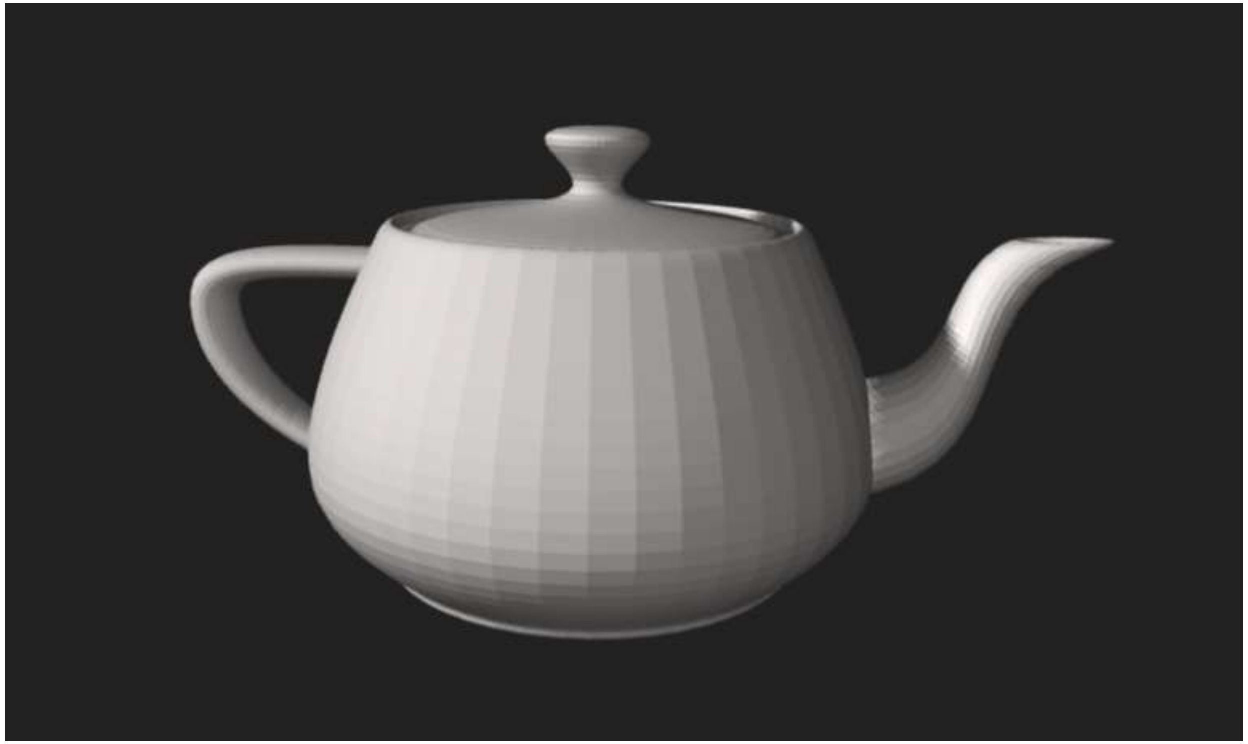
GPU Computing

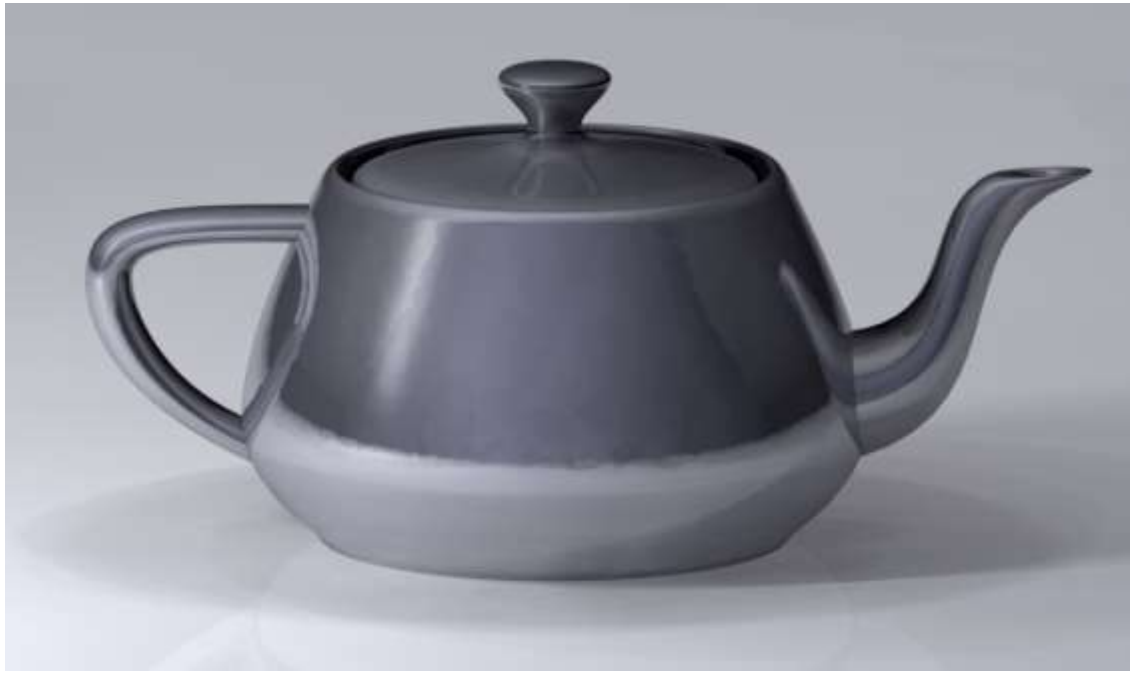
GPU (Graphics Processing Unit)

- Rendering
  - 3D surfaces
  - Textures
  - Lights
  - Views

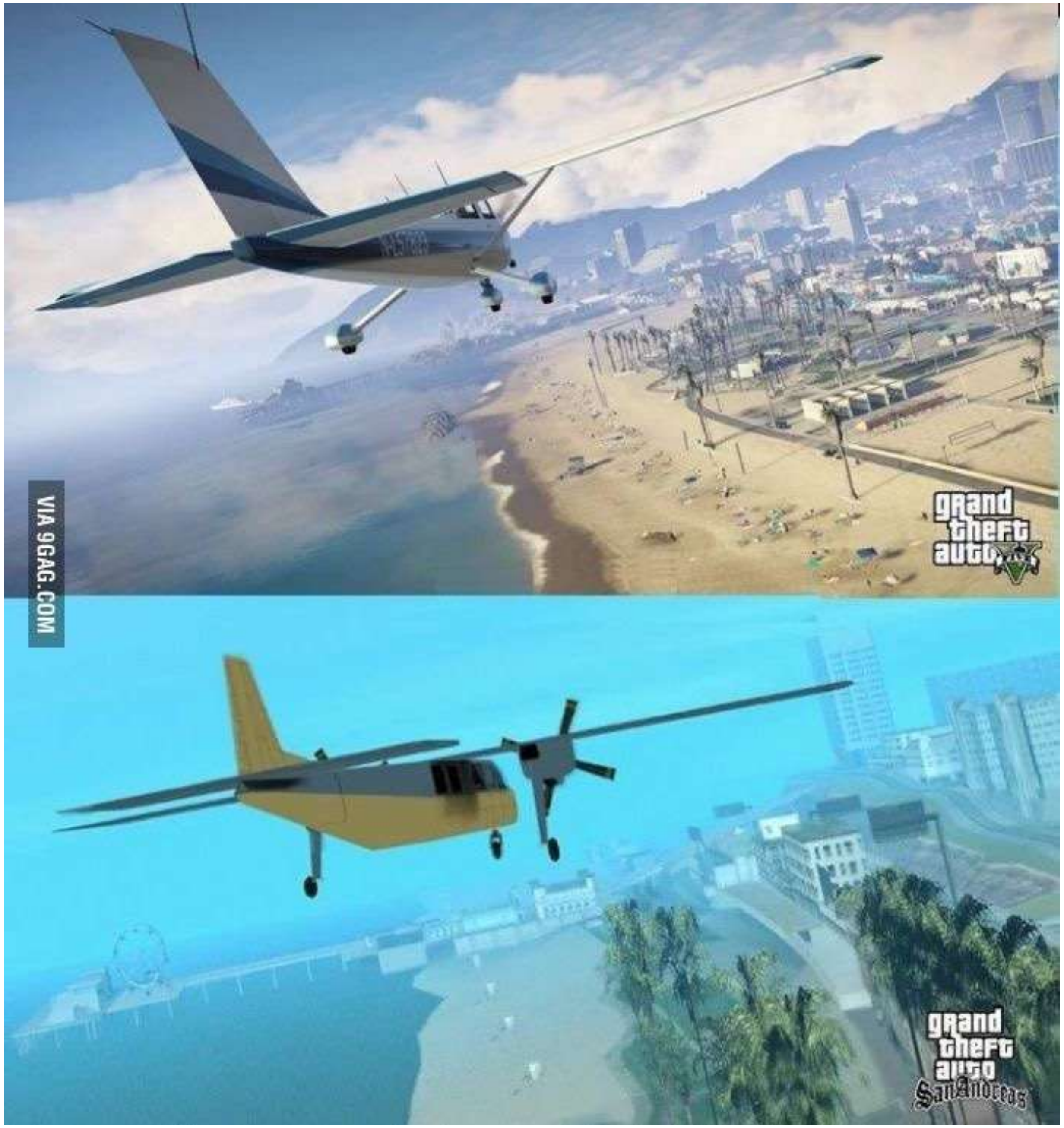


https://en.wikipedia.org/wiki/File:Stanford_bunny_qem.png





https://en.wikipedia.org/wiki/Utah_teapot



https://www.quora.com/What-has-a-better-story-GTA-V-or-GTA-San-Andreas

GPU Rendering

- Direct3D
- OpenGL

- Use primitives to render a frame

GPU ~~Rendering~~ Computing (Before 2007)

- Direct3D
- OpenGL

- Use primitives to render a frame
- Make your 2D array as a frame and call render primitives

GPU Computing (After 2007)

- CUDA (~~Compute Unified Device Architecture~~)
- C/C++, Fortran
- GPGPU (General-Purpose computing on Graphics Processing Units)
- OpenCL

GPU Architecture

- 108 Streaming Multi-processor (SM)
- 40 GB High-Bandwidth Memory (HBM)
  - 1555 GB/sec
  - 6912 FP32 CUDA cores
- 432 Tensor Cores, TensorFloat-32(TF32) Dense Tensor (156 TFLOPs)
- 192KB * 108 L1 Cache
- 40960 KB L2 Cache

GPU Scheduling

- SIMT (Single Instruction Multiple Thread)
- Warp
- Dangerous to implement critical section (Pre Volta)

- Independent Thread Scheduling (After Volta)

CUDA Programming

- Kernel
- Grid
- Block
- Thread
- Warp

- Host Memory
- Device Memory
  - Global Memory
  - Shared Memory

CUDA Programming

```
__global__
void saxpy(int n, float a, float *x, float *y){
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	if (i < n)
		y[i] = a * x[i] + y[i];
}
```

CUDA Programming

```
__global__
void saxpy(int n, float a, float *x, float *y){
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < n)
                y[i] = a * x[i] + y[i];
}
saxpy<<<nB, nT>>>(n, a, x, y);
```

CUDA Programming

device memory

```
__global__
void saxpy(int n, float a, float *x, float *y){
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < n)
                y[i] = a * x[i] + y[i];
}
saxpy<<<nB, nT>>>(n, a, x, y);
```

CUDA Programming

```
__global__
void saxpy(int n, float a, float *x, float *y){
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	if (i < n)
		y[i] = a * x[i] + y[i];
}
saxpy<<<nB, nT>>>(n, a, x, y);
```

device memory

CUDA Programming

device memory

```
__global__
void saxpy(int n, float a, float *x, float *y){
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < n)
                y[i] = a * x[i] + y[i];
}
saxpy<<<nB, nT>>>(n, a, x, y);
```

```
float* x;
cudaMalloc(&x, n * sizeof(float));
```

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

Host Memory vs. Device Memory

- cudaMalloc, cudaFree
- cudaError_t cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)
  - cudaMemcpyHostToHost = 0
    - Host -> Host
  - cudaMemcpyHostToDevice = 1
    - Host -> Device
  - cudaMemcpyDeviceToHost = 2
    - Device -> Host
  - cudaMemcpyDeviceToDevice = 3
    - Device -> Device
  - cudaMemcpyDefault = 4
    - Direction of the transfer is inferred from the pointer values. Requires unified virtual addressing

CUDA Compilation

- nvcc a.cu -o a.out -O3 -Xptxas -O3
- cuda-gdb
  - -g -G (without optimizations)
  - info cuda threads
  - cuda thread 0
- cuda-memcheck
- nvprof
  - nvvp

Scan

- Inclusive scan
- Exclusive scan

- Naïve scan
- Work-efficient scan