



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Programming

Final assignment

Lucia Giorgi

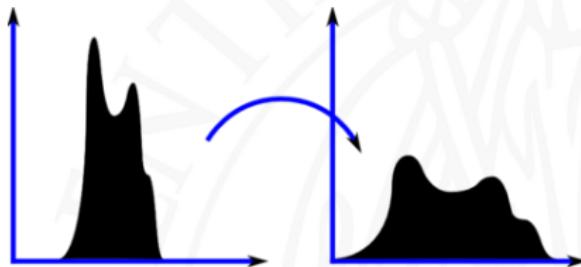
12/01/24



1 Hisotgram Equalization



Histogram equalization is a digital image processing method by which you can calibrate the contrast using the image histogram.



Histogram Equalization



Figure: Image before and after histogram equalization



Histogram Equalization

- Compute the histogram: for every possible value of a pixel (0 - 255) count how many pixels have that value $p(v)$ for $v \in [0, L - 1]$, $L = 256$



Histogram Equalization

- Compute the histogram: for every possible value of a pixel (0 - 255) count how many pixels have that value $p(v)$ for $v \in [0, L - 1]$, $L = 256$
- Compute the CDF:

$$\text{cdf}(v) = \sum_{k=1}^v p(k)$$



Histogram Equalization

- Compute the histogram: for every possible value of a pixel (0 - 255) count how many pixels have that value $p(v)$ for $v \in [0, L - 1]$, $L = 256$
- Compute the CDF:

$$\text{cdf}(v) = \sum_{k=1}^v p(k)$$

- For every image pixel compute the new value:

$$h(v) = \frac{\text{cdf}(v) - \text{cdf}_{\min}}{MN - \text{cdf}_{\min}} \cdot (L - 1)$$



Histogram Equalization

RGB → HSV → histogram equalization on V → RGB

Three kernels:

- `histogram()`
generates the histogram from the input image
- `Brent_Kung_scan_kernel()`
calculates the CDF
- `equalizer()`
generates the output image



Histogram

- $M \cdot N$ threads
- $\text{BLOCK_DIM_X} * \text{BLOCK_DIM_Y}$ threads per block
- **shared memory:** $256 * \text{sizeof(int)}$ bytes

Shared memory $256 * \text{sizeof(int)}$

0	1	2	3	...	255
1	0	0	0	0	0

Global memory

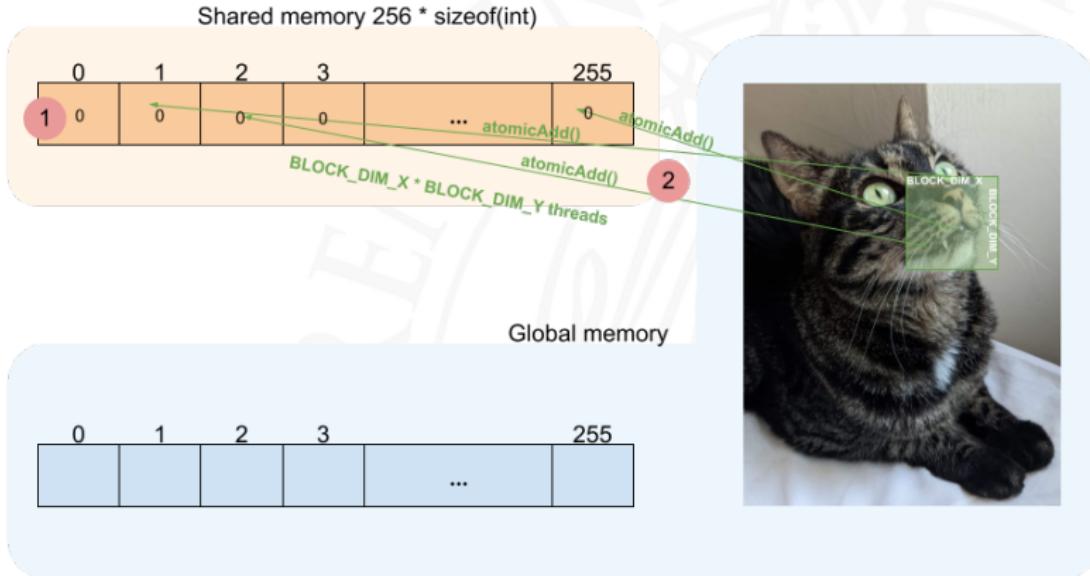
0	1	2	3	...	255





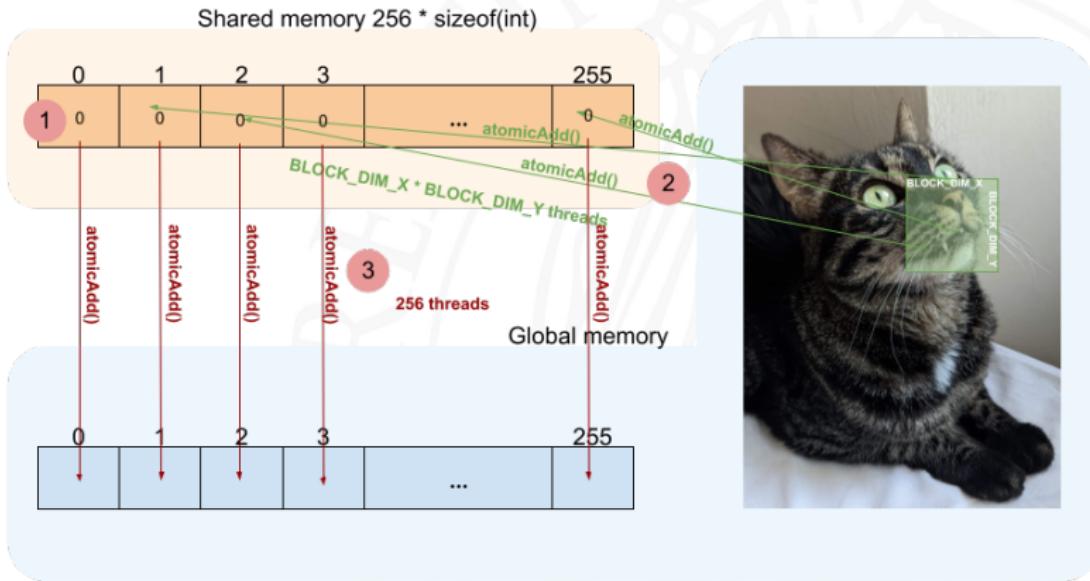
Histogram

- $M \cdot N$ threads
- $\text{BLOCK_DIM_X} * \text{BLOCK_DIM_Y}$ threads per block
- **shared memory:** $256 * \text{sizeof(int)}$ bytes



Histogram

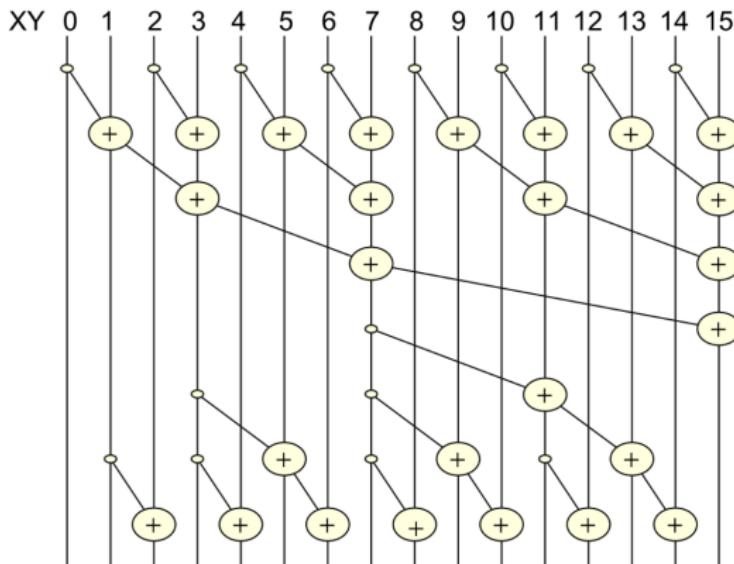
- $M \cdot N$ threads
- $\text{BLOCK_DIM_X} * \text{BLOCK_DIM_Y}$ threads per block
- **shared memory:** $256 * \text{sizeof(int)}$ bytes



Histogram

```
__global__ void histogram(const unsigned char* image, int* hist, int width, int height) {
    __shared__ int HIST[HIST_SIZE];
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int t_id = tx + ty * width;
    int t_id_block = threadIdx.x + threadIdx.y * BLOCK_DIM_X;
    if (t_id_block < HIST_SIZE) { // shared memory initialization
        HIST[t_id_block] = 0;
    }
    __syncthreads();
    if (tx < width && ty < height) {
        atomicAdd(&(HIST[image[t_id]]), 1); // local histogram
    }
    __syncthreads();
    if (t_id_block < HIST_SIZE) {
        atomicAdd(&(hist[t_id_block]), HIST[t_id_block]); // global histogram
    }
}
```

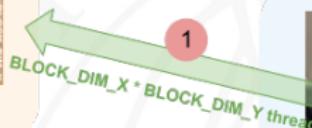
Based on the Brent–Kung adder design



Equalizer

- $M \cdot N$ threads
- $\text{BLOCK_DIM_X} * \text{BLOCK_DIM_Y}$ threads per block
- **shared memory:** $\text{BLOCK_DIM_X} * \text{BLOCK_DIM_Y} * \text{sizeof(int)}$ bytes

Shared memory
 $\text{BLOCK_DIM_X} * \text{BLOCK_DIM_Y} * \text{sizeof(int)}$



Global memory



Constant memory

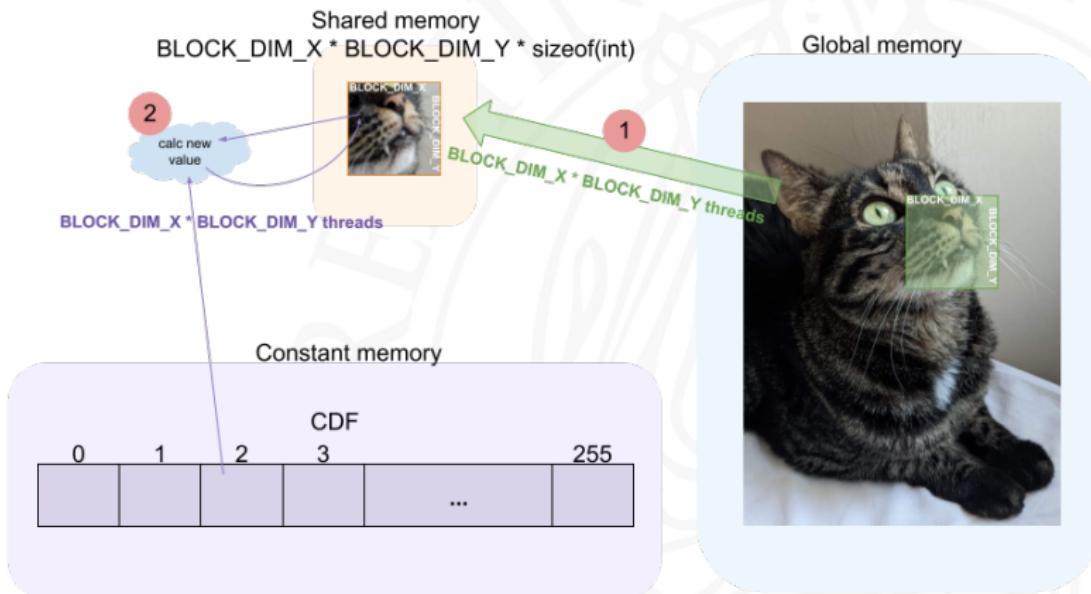
CDF

0 1 2 3 ... 255



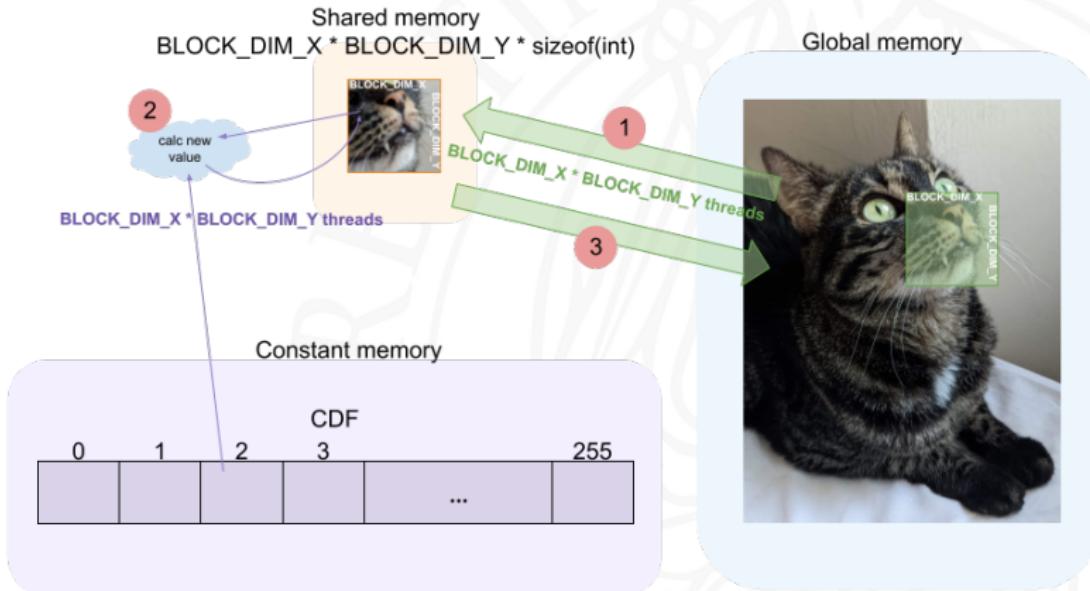
Equalizer

- $M \cdot N$ threads
- $\text{BLOCK_DIM_X} * \text{BLOCK_DIM_Y}$ threads per block
- **shared memory:** $\text{BLOCK_DIM_X} * \text{BLOCK_DIM_Y} * \text{sizeof(int)}$ bytes



Equalizer

- $M \cdot N$ threads
- $\text{BLOCK_DIM_X} * \text{BLOCK_DIM_Y}$ threads per block
- **shared memory:** $\text{BLOCK_DIM_X} * \text{BLOCK_DIM_Y} * \text{sizeof(int)}$ bytes





Equalizer

```
__constant__ int d_CDF[HIST_SIZE];
__global__ void equalizer(unsigned char* image, int cdf_val_min, int width, int height) {
    __shared__ unsigned char IMG[BLOCK_DIM_X * BLOCK_DIM_Y];
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int t_id = tx + ty * width;
    int t_id_block = threadIdx.x + threadIdx.y * BLOCK_DIM_X;
    if (tx < width && ty < height) {
        IMG[t_id_block] = image[t_id];
    }
    __syncthreads();
    if (tx < width && ty < height) {
        IMG[t_id_block] = long(d_CDF[IMG[t_id_block]] - cdf_val_min) * (HIST_SIZE - 1) /
                           (width * height - cdf_val_min);
    }
    __syncthreads();
    if (tx < width && ty < height) {
        image[t_id] = IMG[t_id_block];
    }
}
```

Experiments

- Multiple image sizes
 - 10 images per size (5 vertical + 5 horizontal)

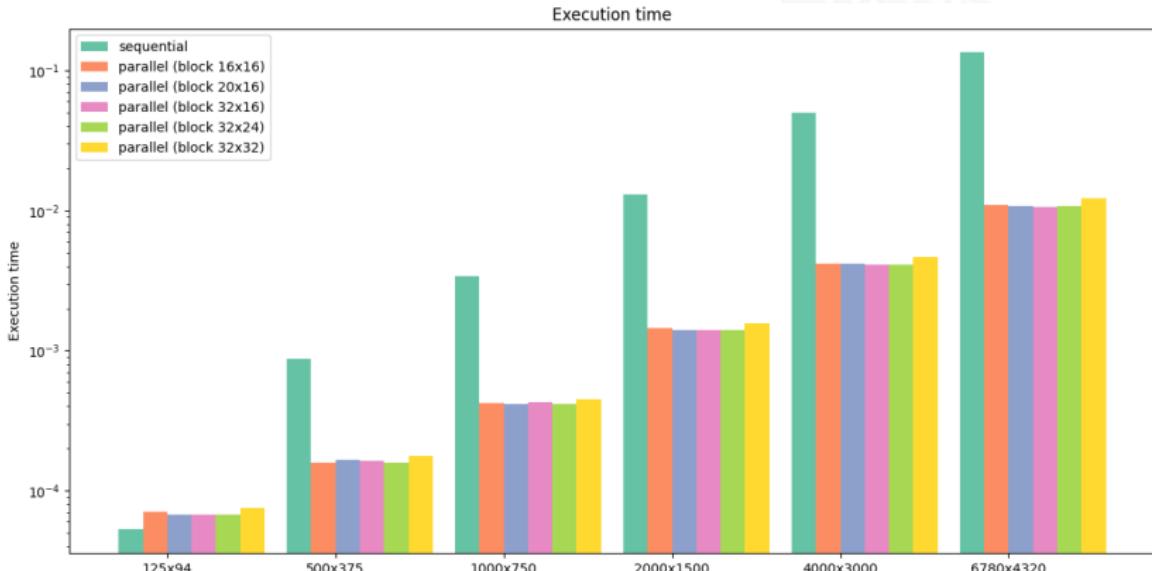
Experiments

- Multiple image sizes
 - 10 images per size (5 vertical + 5 horizontal)
- Multiple block sizes
 - ($\min \text{BLOCK_DIM_X} * \text{BLOCK_DIM_Y} = 256$)

Experiments

- Multiple image sizes
 - 10 images per size (5 vertical + 5 horizontal)
- Multiple block sizes
 - $(\min \text{BLOCK_DIM_X} * \text{BLOCK_DIM_Y} = 256)$
- "Dummy" image for initialization
 - NVIDIA Nsight Systems (profiling)

Results



Results

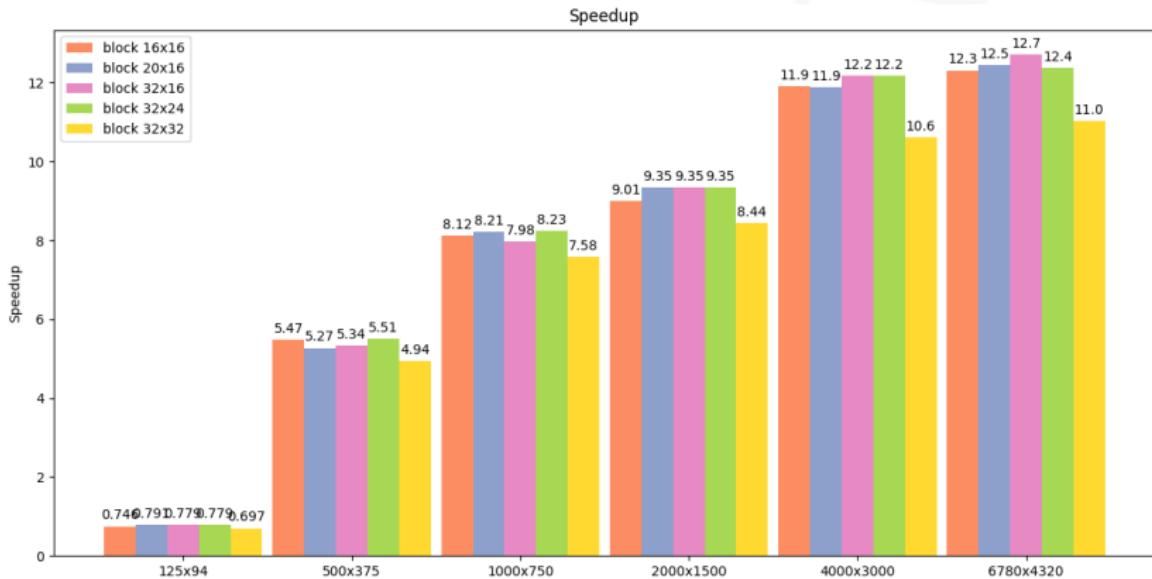


Figure: Speedup

Occupancy

In order to use

NVIDIA Nsight Compute - Occupancy Calculator

- `__host__ cudaError_t cudaGetDeviceProperties (cudaDeviceProp* prop, int device)`
for info about the GPU
 - total shared memory per multiprocessor
- `--ptxas-options=-v` flag in nvcc
for info about the kernels
 - number of registers
- NVIDIA website for the compute capability

Occupancy

- 67% for $32 \times 32 = 1024$ block size
- 83% for $20 \times 16 = 320$ block size
- 100% for the other block sizes



Occupancy

- 67% for $32 \times 32 = 1024$ block size
- 83% for $20 \times 16 = 320$ block size
- 100% for the other block sizes

Block size $32 \times 32 = 1024$:

- Maximum number of threads per SM: 1536
- Threads per SM for: 1024
- $1024/1536 = 0.67$

Equalizer

The screenshot shows the NVIDIA Nsight Compute application window. The top menu bar includes File, Connection, Debug, Profile, Tools, Window, Help, File, Connect, X, Terminate, Profile Kernel, Metrics, Metrics Details, and Baseline. The left sidebar has Project Explorer, Search project, and Default Project sections. The main area displays occupancy analysis for a CUDA kernel:

- Compute Capability:** 8.6
- Threads Per Block:** 1024
- Registers Per Thread:** 27
- Global Load Cache Mode:** L1+L2 (iso)
- User Shared Memory Per Block (bytes):** 1024

Below these settings are three tabs: Tables, Graphs, and GPU Data. The GPU Data tab is selected, showing the following data:

Occupancy Data:		Physical Limit of GPU (8.6):	
Property	Value	Property	Limit
Active Threads per Multiprocessor	1024	Threads per Warp	32
Active Warps per Multiprocessor	32	Max Threads per Multiprocessor	44
Active Thread Blocks per Multiprocessor	1	Max Thread Blocks per Multiprocessor	16
Occupancy of each Multiprocessor	67.7%	Max Threads per Multiprocessor	1024
		Maximum Thread Block Size	1024
		Registers per Multiprocessor	65536
		Max Registers per Thread Block	255
		Max Registers per Thread	255
		Shared Memory per Multiprocessor (bytes)	102400
		Max Shared Memory per Block	102400
		Register Allocation Unit Size	56
		Warp Allocation Granularity	warp
		Shared Memory Allocation Unit Size	128
		Warp Allocation Granularity	4
		Shared Memory Per Block (bytes) (CUDA runtime use)	1024

Below the physical limit table are sections for Allocated Resources and Occupancy Limiters.

Allocated Resources:

Resources	Per Block	Limit Per SM	Allocatable Blocks Per SM
Warp (Threads Per Block / Threads Per Warp)	32	48	1
Registers (Warp limit per SM due to per-warp reg count)	32	64	2
Shared Memory (Bytes)	2048	102400	50

Occupancy Limiters:

Limited By	Blocks per SM	Warp Per Block	Warp Per SM
Max Warps or Max Blocks per Multiprocessor	1	32	32
Registers per Multiprocessor	2		
Shared Memory per Multiprocessor	50		

The bottom status bar shows the system tray with icons for battery, signal, volume, and time (mer 10 gen 22:29).