Parallel Programming
Final assignment

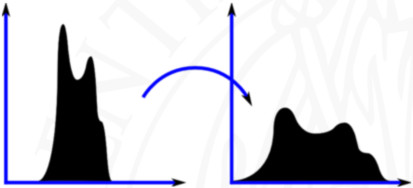
Lucia Giorgi

12/01/24

Index

1. Histogram Equalization
2. Cuda
3. Experiments and results

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

UNIVERSITÀ DEGLI STUDI FIRENZE

Histogram Equalization

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

UNIVERSITÀ DEGLI STUDI FIRENZE

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:

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

UNIVERSITÀ DEGLI STUDI FIRENZE

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:

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

Histogram Equalization

RGB $\rightarrow$ HSV $\rightarrow$ histogram equalization on V $\rightarrow$ RGB

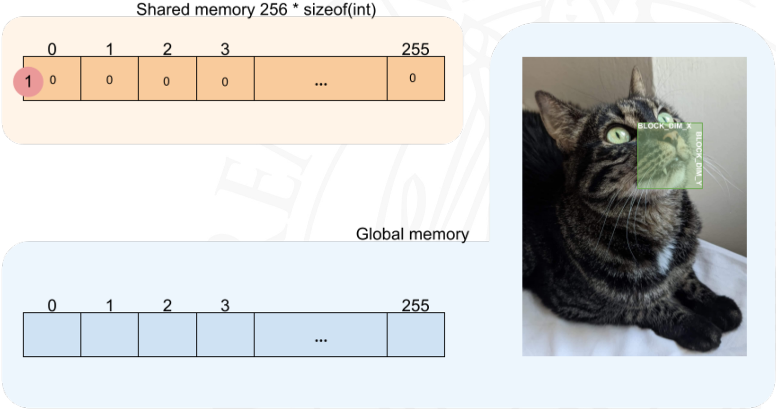
Cuda

Three kernels:

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

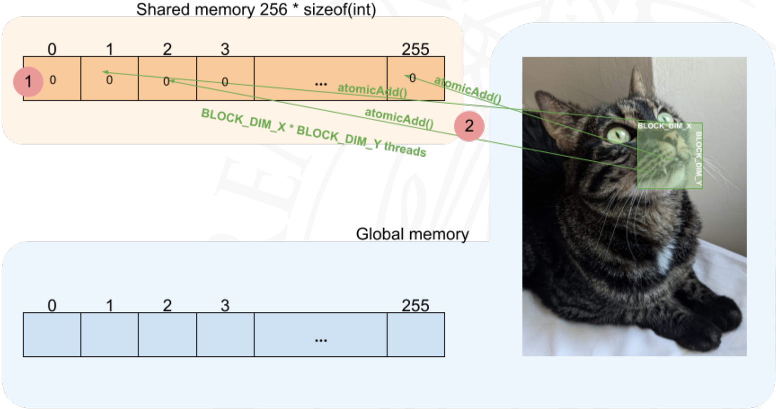
Histogram

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



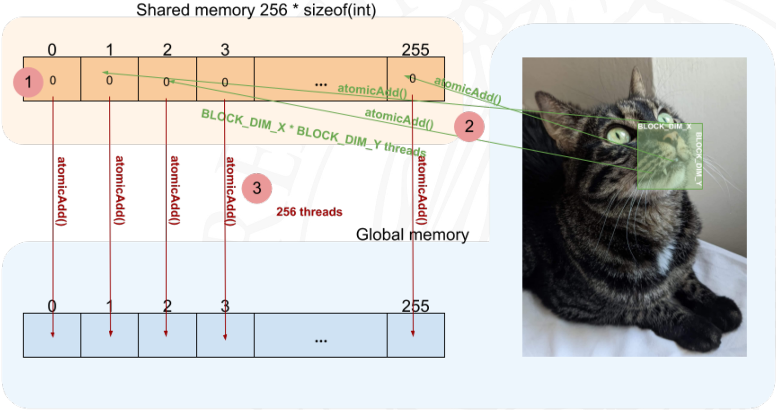
Histogram

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



Histogram

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



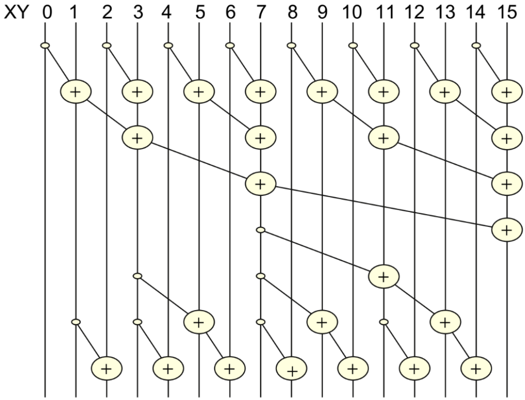
UNIVERSITÀ DEGLI STUDI FIRENZE

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
    }
}
```

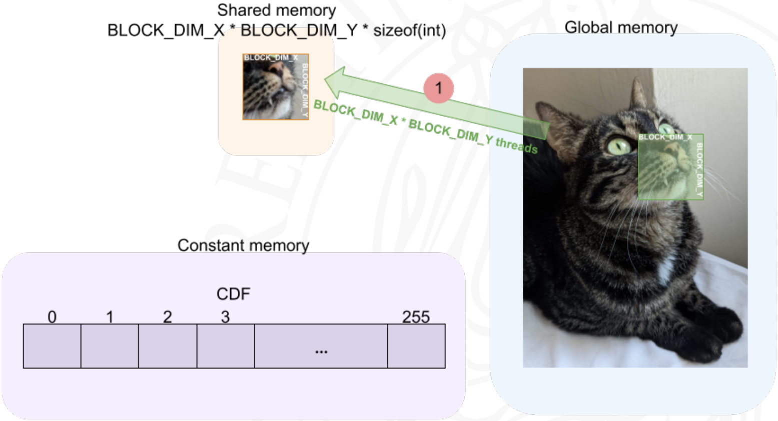
Scanner

Based on the Brent–Kung adder design



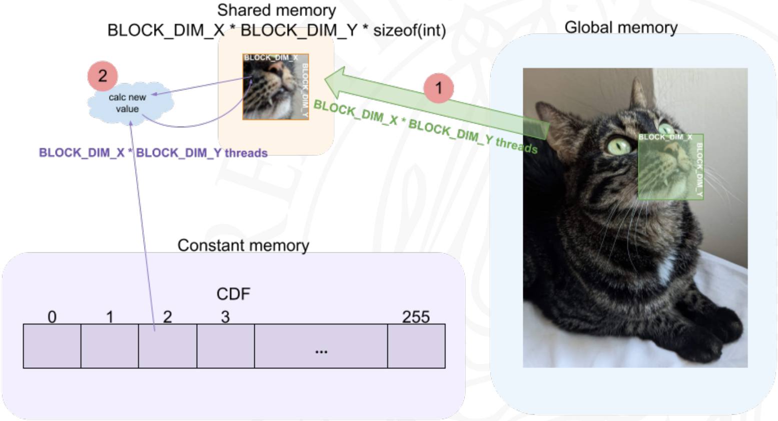
Equalizer

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



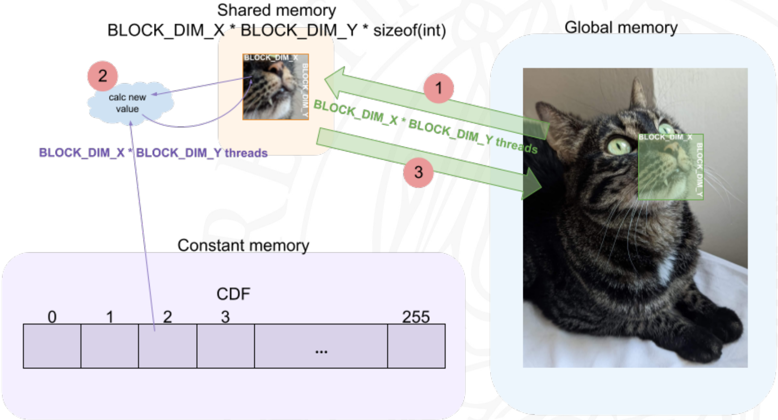
Equalizer

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



Equalizer

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



UNIVERSITÀ
DEGLI STUDI
FIRENZE

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 `BLOCK_DIM_X * BLOCK_DIM_Y` = 256)

Experiments

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

Results

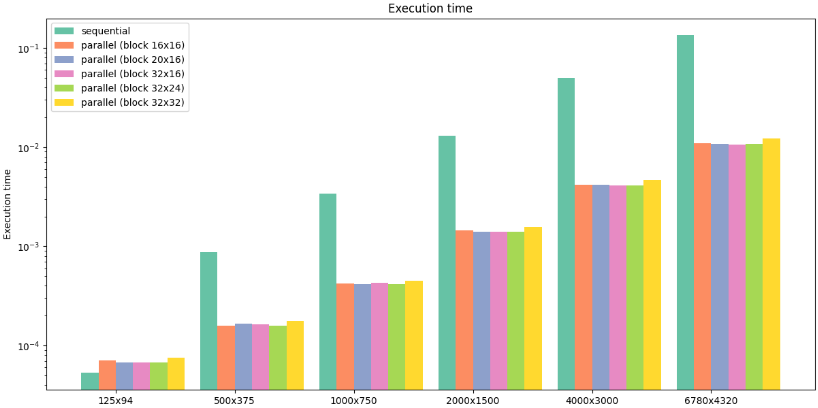


Image size	execution time sequential (s)	execution time 16x16 (s)	execution time 20x16 (s)	execution time 32x16 (s)	execution time 32x24 (s)	execution time 32x32 (s)
125x94	5.3e-05	7.1e-05	6.7e-05	6.8e-05	6.8e-05	7.6e-05
500x375	0.00087	0.000159	0.000165	0.000163	0.000158	0.000176
1000x750	0.003409	0.00042	0.000415	0.000427	0.000414	0.00045
2000x1500	0.013126	0.001457	0.001404	0.001404	0.001404	0.001556
4000x3000	0.049783	0.004177	0.00419	0.004084	0.004086	0.004694
6780x4320	0.134443	0.010917	0.010793	0.010579	0.010857	0.012192

Results

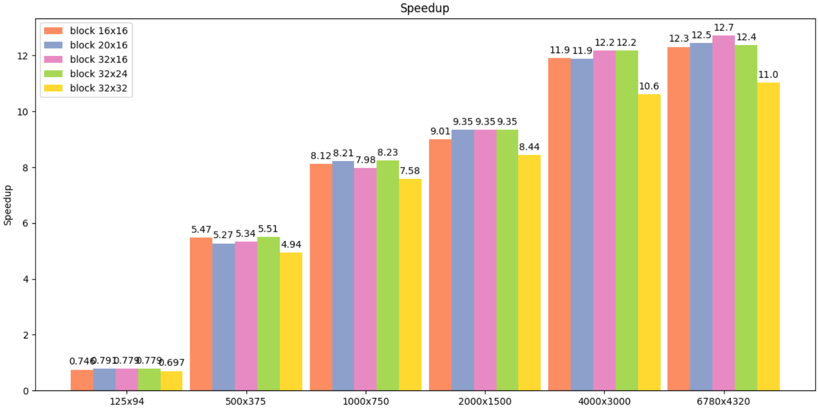


Figure: Speedup

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

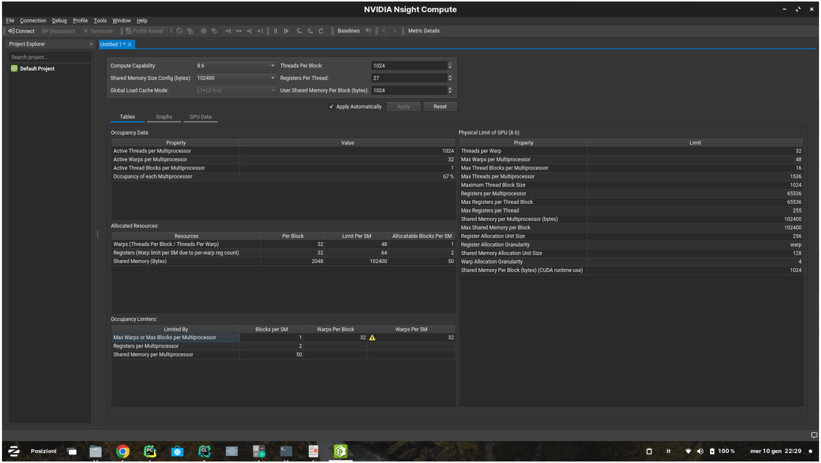
Occupancy

- 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


NVIDIA Nsight Compute
File Connection Debug Profile Tools Window Help
Connect Disconnect Terminate Profile Kernel Baselines Metric Details
Project Explorer
Search project...
Default Project
Untitled 1 *
Compute Capability: 8.6
Shared Memory Size Config (bytes): 102400
Global Load Cache Mode: L1+L2 (ca)
Threads Per Block: 1024
Registers Per Thread: 27
User Shared Memory Per Block (bytes): 1024
Apply Automatically Apply Reset
Tables Graphs GPU Data
Occupancy Data:
Property Value
Active Threads per Multiprocessor 1024
Active Warps per Multiprocessor 32
Active Thread Blocks per Multiprocessor 1
Occupancy of each Multiprocessor 67 %
Allocated Resources:
Resources Per Block Limit Per SM Allocatable Blocks Per SM
Warps (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 Warps Per Block Warps Per SM
Max Warps or Max Blocks per Multiprocessor 1 32 32
Registers per Multiprocessor 2
Shared Memory per Multiprocessor 50
Physical Limit of GPU (8.6):
Property Limit
Threads per Warp 32
Max Warps per Multiprocessor 48
Max Thread Blocks per Multiprocessor 16
Max Threads per Multiprocessor 1536
Maximum Thread Block Size 1024
Registers per Multiprocessor 65536
Max Registers per Thread Block 65536
Max Registers per Thread 255
Shared Memory per Multiprocessor (bytes) 102400
Max Shared Memory per Block 102400
Register Allocation Unit Size 256
Register Allocation Granularity warp
Shared Memory Allocation Unit Size 128
Warp Allocation Granularity 4
Shared Memory Per Block (bytes) (CUDA runtime use) 1024
Posizioni
it 100 % mer 10 gen 22:29