PC-2024/25 Histogram Equalization

Alessio Bugetti
7132744
alessio.bugetti@edu.unifi.it

Abstract

This report focuses on the implementation and analysis of histogram equalization in both sequential and parallel contexts. The sequential version was implemented as a baseline, while the parallel version leverages CUDA to exploit GPU acceleration. Three distinct scan algorithms—Kogge-Stone, Kogge-Stone with Double Buffering, and Brent-Kung—were employed in the parallel implementation to optimize performance. Execution times were measured across varying image dimensions and block sizes. Speedup factors were calculated by comparing parallel execution times with the sequential baseline. This report underscores the efficacy of CUDA in accelerating computationally intensive image processing tasks.

1. Introduction

The goal of this project is to implement histogram equalization for image processing both sequentially on the CPU and in parallel on the GPU using CUDA, while analyzing the resulting speedup. Histogram Equalization is a fundamental technique in image processing used to enhance the contrast of an image. This method is particularly effective for images where the intensity levels of pixels are concentrated within a limited range, which often results in poor visual contrast and reduced perceptibility of details. By redistributing these intensity levels over the entire available range, histogram equalization improves the visual quality of the image and makes subtle details more distinguishable.

The process begins by analyzing the intensity distribution of the image, which is represented by its histogram. This histogram provides a statistical overview of how pixel intensities are distributed across the image. In cases where the histogram shows a narrow concentration of values, the image is likely to appear washed out or underexposed. Histogram equalization addresses this issue by transforming the intensity values based on the cumulative distribution function (CDF) derived from the histogram. The CDF acts as a mapping function, which reallocates pixel intensities in a way that spreads them more evenly across the entire intensity range, often from 0 to 255 for standard grayscale images.

This transformation effectively enhances the contrast by ensuring that all intensity levels are utilized more uniformly. The result is an image with improved brightness and detail visibility. For example, in an image where the pixel intensities were originally confined to a range between 50 and 100, the equalization process would stretch these values to span the entire range from 0 to 255. This stretching of intensities not only improves the overall visual quality but also highlights previously indistinct features in the image.

Histogram equalization is computationally simple and is widely applied in various domains. However, it is not without limitations. The global nature of the technique may introduce noise or artifacts in cases where the original histogram is already well-distributed. Furthermore, it may not account for local variations in intensity, making it less effective for images with region-specific contrast issues.

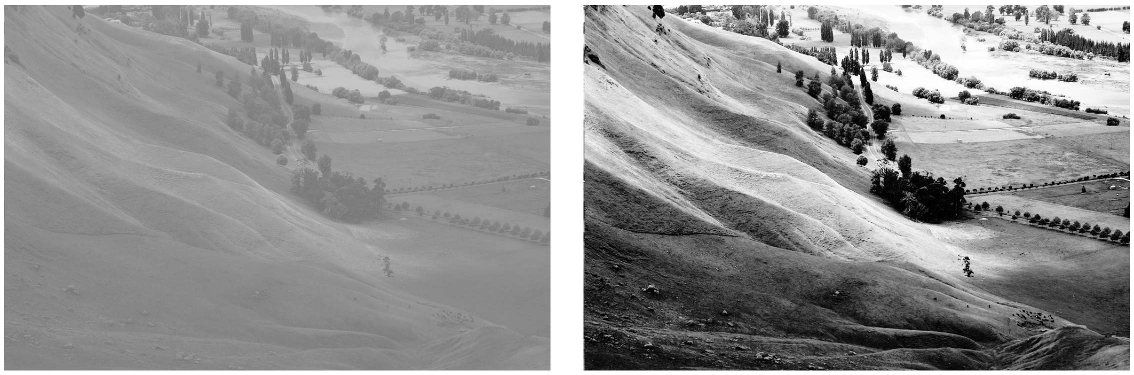
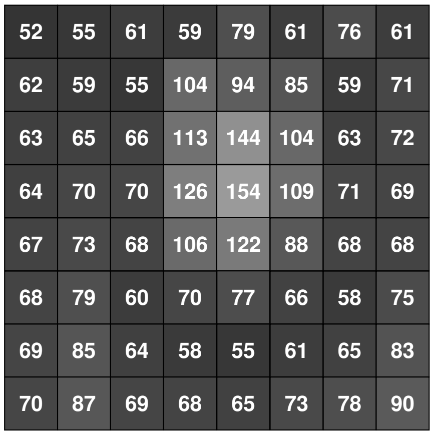


Figure 1: A concrete example of histogram equalization application. On the left, the original image, on the right, the equalized image.

1.1. Histogram Equalization Execution



52	55	61	59	79	61	76	61
62	59	55	104	94	85	59	71
63	65	66	113	144	104	63	72
64	70	70	126	154	109	71	69
67	73	68	106	122	88	68	68
68	79	60	70	77	66	58	75
69	85	64	58	55	61	65	83
70	87	69	68	65	73	78	90

Figure 2: The original 8-bit grayscale image.

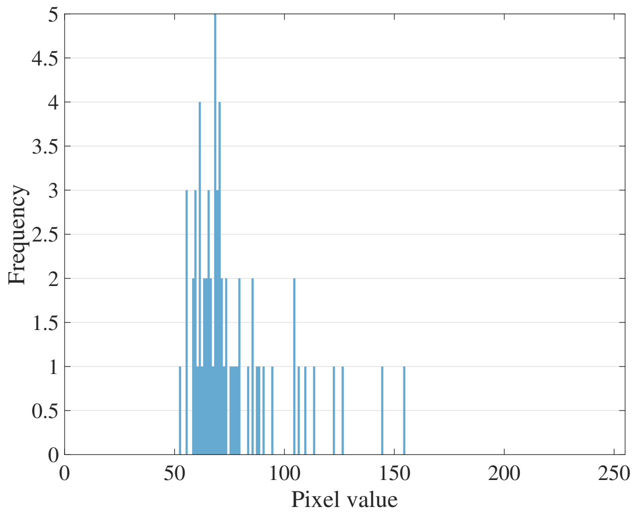


Figure 3: Histogram of the 8x8 grayscale image.

Consider the 8-bit grayscale image represented in Figure 2. The first step in performing histogram equalization on this image is to define its histogram, which is shown in Figure 3. One can observe that the pixel values are concentrated between $[52, 154]$, and the goal is to stretch these values to span the entire range $[0, 255]$. From its histogram, we can obtain its cumulative distribution function (CDF), which is normalized using the following formula:

$$cdf(v) = \left\lfloor \frac{cdf(v) - cdf_{min}}{(M \times N) - cdf_{min}} \times (L - 1) \right\rceil \qquad (1)$$

where v is a cumulative bin of the CDF, cdf_{min} is the value of the first non-zero cumulative bin of the CDF, $M \times N$ is the total number of pixels in the image, and L is the number of grayscale levels used. In this case $M \times N = 64$ and $L = 256$.

Once the CDF is normalized, the new values for the pixels of the image are obtained by mapping them to the normalized CDF as follows:

$$image(pixel) = cdf(image(pixel))$$

In this case, the histogram corresponding to the equalized image can be observed in Figure 4, and the equalized image itself is directly shown in Figure 5.

2. Project Structure

The project is structured into two main implementations: a sequential version and a parallel

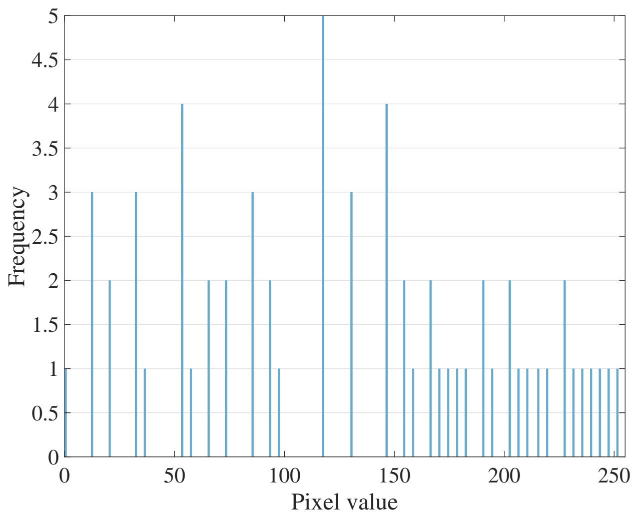
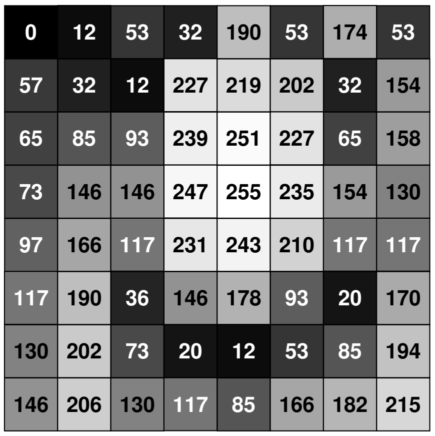


Figure 4: Histogram of the equalized 8x8 grayscale image.



0	12	53	32	190	53	174	53
57	32	12	227	219	202	32	154
65	85	93	239	251	227	65	158
73	146	146	247	255	235	154	130
97	166	117	231	243	210	117	117
117	190	36	146	178	93	20	170
130	202	73	20	12	53	85	194
146	206	130	117	85	166	182	215

Figure 5: The equalized 8-bit grayscale image.

version using CUDA. The sequential implementation serves as a baseline for performance comparison, while the parallel version leverages GPU acceleration to enhance efficiency.

The script `histequalizer` has been developed for the implementation in C++ and allows you to compile the project using Ninja with:

```
./histequalizer build
```

and to run the benchmarks defined in `main.cu` with:

```
./histequalizer run
```

Instead, to run the benchmarks for the Python implementation:

```
python main.py
```

2.1. Sequential Implementation

Below is the pseudocode for the sequential implementation:

Algorithm 1 Histogram Equalization

```
Require: $image$: grayscale image of size $M \times N$ with intensity range $[0, L-1]$
Ensure: $equalized\_image$: output image of the same size
1: Initialize $hist \leftarrow$ array of size $L$ filled with zeros
2: for each pixel $p$ in $image$ do
3:     $hist[p] \leftarrow hist[p] + 1$
4: end for
5: Initialize $cdf \leftarrow$ array of size $L$ filled with zeros
6: $cdf[0] \leftarrow hist[0]$
7: for $i = 1$ to $L-1$ do
8:     $cdf[i] \leftarrow cdf[i-1] + hist[i]$
9: end for
10: $cdf\_min \leftarrow$ first nonzero element in $cdf$
11: $total\_pixels \leftarrow M \times N$
12: for $i = 0$ to $L-1$ do
13:     $cdf[i] \leftarrow \text{round}\left(\frac{cdf[i]-cdf\_min}{total\_pixels-cdf\_min} \times (L-1)\right)$
14: end for
15: for each pixel $p$ in $image$ do
16:     $equalized\_image[p] \leftarrow cdf[image[p]]$
17: end for
18: return $equalized\_image$
```

The concrete implementation in C++ used is as follows:

```
void SequentialHistogramEqualization(
  const unsigned char *input,
  unsigned char *output,
  const unsigned int pixelCount) {
  if (input == nullptr || output == nullptr ||
      pixelCount == 0) {
    return;
  }

  unsigned int histogram[NUM_BINS] = {0};
  unsigned int cdf[NUM_BINS] = {0};

  bool cdfMinIsSet = false;
  unsigned int cdfMin;

  for (unsigned int i = 0; i < pixelCount; i++) {
    histogram[input[i]]++;
  }

  for (unsigned int i = 0; i < NUM_BINS; i++) {
    cdf[i] = histogram[i];
    if (i > 0) {
      cdf[i] += cdf[i - 1];
    }
    if (!cdfMinIsSet && cdf[i] > 0) {
      cdfMin = cdf[i];
      cdfMinIsSet = true;
    }
  }

  if (pixelCount == cdfMin) {
    for (unsigned int i = 0; i < pixelCount;
        i++) {
      output[i] = input[i];
    }
    return;
  }

  for (unsigned int &value : cdf) {
    value = ((value - cdfMin) * (NUM_BINS - 1) +
        (pixelCount - cdfMin) / 2) /
            (pixelCount - cdfMin);
  }

  for (unsigned int i = 0; i < pixelCount; i++) {
    output[i] = static_cast<unsigned
        char>(cdf[input[i]]);
  }
}
```

First, an array of 256 unsigned integers is created, one for each possible pixel value in the image, which represents the histogram and is initialized with all zeros.

Next, the input array, which contains the pixel values of the image, is iterated and for each pixel value the corresponding cell in the $histogram$ array is incremented by 1.

To obtain the CDF, a for loop is structured to compute it in a single pass over the array representing the histogram. The cdf_{min} is also tracked, which is required in the next step.

For the normalization of the CDF, another pass is made through the array representing the CDF, and the normalization formula is the one highlighted in Equation 1.

Finally, the output image is generated by mapping the value of each pixel from the input image to the newly normalized CDF, which requires iterating through a for loop over all the pixels.

It is clear that, in general, the bottlenecks of the proposed sequential algorithm are the loops that iterate over all the pixels in the image, since they are typically numerically much more numerous than the number of possible grayscale values chosen, i.e., 256. Therefore, the performance gain in terms of execution time and speedup is expected mainly from effectively parallelizing the histogram creation and the mapping onto the normalized CDF.

A similar sequential implementation has also been developed in Python.

2.2. Parallel Implementation with CUDA

In this project, histogram equalization was implemented and evaluated with three distinct scan algorithms: Kogge-Stone, Kogge-Stone with double buffering, and Brent-Kung, using CUDA to leverage the inherent parallelism of GPUs.

The implementation begins by computing the histogram of the input image through the `CalculateHistogram` kernel:

```
__global__ void CalculateHistogram(
  const unsigned char *input,
  unsigned int *histogram,
  const unsigned int pixelCount) {
  __shared__ unsigned int cache[NUM_BINS];
  for (unsigned int bin = threadIdx.x; bin <
      NUM_BINS; bin += blockDim.x) {
    cache[bin] = 0;
  }
  __syncthreads();

  unsigned int accumulator = 0;
  unsigned int prevBin = NUM_BINS;

  const unsigned int tid = blockIdx.x *
      blockDim.x + threadIdx.x;

  for (unsigned int index = tid; index <
      pixelCount;
       index += blockDim.x * gridDim.x) {
    unsigned int currBin = input[index];
    if (currBin != prevBin) {
      if (prevBin != NUM_BINS) {
        atomicAdd(&cache[prevBin], accumulator);
      }
      accumulator = 1;
      prevBin = currBin;
    } else {
      accumulator++;
    }
  }
  if (accumulator > 0) {
    atomicAdd(&cache[prevBin], accumulator);
  }
  __syncthreads();

  for (unsigned int bin = threadIdx.x; bin <
      NUM_BINS;
       bin += blockDim.x) {
    unsigned int binValue = cache[bin];
    if (binValue > 0) {
      atomicAdd(&histogram[bin], binValue);
    }
  }
}
```

When observing the implementation, it is important to emphasize the use of shared memory, which is shared among the threads of the block and is used to temporarily store the histogram values. This helps reduce the number of accesses to global memory, which is slower compared to shared memory. The first `__syncthreads()` ensures that all threads in the block have completed the shared memory initialization before proceeding.

To avoid losing the benefits of shared memory usage, thread coarsening is applied. This means reducing the number of blocks and assigning each thread multiple input elements using interleaved partitioning.

For example, in the second for loop, each thread, in its first iteration, accesses the input array using its global index and updates the `cache` array, which is stored in shared memory. This update is performed using the `atomicAdd` function, which ensures atomic operations to prevent race conditions. Specifically, the update is performed using the Aggregation technique[2], which is based on the idea that each thread should aggregate consecutive updates into a single update if they are updating the same element of the histogram. This aggregation reduces the number of atomic operations, improving the effective throughput of the computation. This is useful when there is a large concentration of identical data values in localized areas.

As a result, in the first iteration all threads collectively process the first `blockDim.x * gridDim.x` elements of the input. In the second iteration, all threads increment their indices by `blockDim.x * gridDim.x` and process the next section of `blockDim.x * gridDim.x` elements. When a thread's index exceeds the valid range of the input buffer, it has finished processing its assigned partition and exits the loop.

The second `__syncthreads()` ensures that all threads have completed updating the `cache`. Finally, the values computed in `cache` are added to the global histogram using `atomicAdd`.

Given a certain number of threads per block, the number of blocks required to cover the entire image is defined as:

$$N_{blocks} = \left\lceil \frac{\text{num. pixel}}{\text{num. threads per block}} \right\rceil \tag{2}$$

This kernel is executed with a 1D grid composed of a number of blocks equal to $\frac{1}{4}N_{blocks}$.

Subsequently, an inclusive scan algorithm is executed to obtain the CDF from the histogram. Three different algorithms have been implemented: Kogge-Stone, Kogge-Stone Double Buffer, and Brent-Kung. Their specific operation and implementations are discussed in Section 2.2.1.

Regardless of the chosen scan algorithm, once the associated CDF is obtained, it is normalized through the `NormalizeCdf` kernel:

```
__global__ void NormalizeCdf(
  unsigned int *cdf,
  const unsigned int pixelCount) {
  __shared__ unsigned int cdfMinIndex;

  const unsigned int tid = threadIdx.x;

  if (tid == 0) {
    cdfMinIndex = 256;
  }
  __syncthreads();

  if (cdf[tid] != 0) {
    atomicMin(&cdfMinIndex, tid);
  }
  __syncthreads();

  __shared__ unsigned int cdfMin;

    if (tid == 0) {
    cdfMin = cdf[cdfMinIndex];
  }
  __syncthreads();

  if (tid < NUM_BINS) {
    cdf[tid] =
        ((cdf[tid] - cdfMin) * (NUM_BINS - 1) +
            (pixelCount - cdfMin) / 2) /
        (pixelCount - cdfMin);
  }
}
```

This is executed with a 1D grid composed of a single block containing 256 threads, so that each bin is processed by a single thread.

Finally, the `EqualizeHistogram` kernel applies the normalized CDF to adjust the input image's intensity levels:

```
__global__ void EqualizeHistogram(
  unsigned char *output,
  const unsigned char *input,
  const unsigned int *cdf,
  const unsigned int pixelCount) {
  __shared__ unsigned int cache[NUM_BINS];
  for (unsigned int bin = threadIdx.x; bin <
      NUM_BINS;
       bin += blockDim.x) {
    cache[bin] = cdf[bin];
  }
  __syncthreads();

  const unsigned int tid = blockIdx.x *
      blockDim.x + threadIdx.x;

  for (unsigned int index = tid; index <
      pixelCount;
       index += blockDim.x * gridDim.x) {
    output[index] = cache[input[index]];
  }
}
```

It can be observed that in this implementation shared memory is utilized to store the normalized CDF that has just been calculated. Additionally, as in the `CalculateHistogram` kernel, multiple input elements are assigned to each thread using interleaved partitioning.

The mapping of the image to the normalized CDF is executed with a 1D grid composed of a number of blocks equal to $\frac{1}{4}N_{blocks}$.

2.2.1 Scan Algorithms

The goal of the implemented Parallel Inclusive Scan Algorithms is to take an input array:

$$A = [a_1, a_2, \ldots, a_{n-1}]$$

and return an output array S defined as:

$$S = [s_1, s_2, \ldots, s_{n-1}]$$

such that $s_i = a_0 + a_1 + \ldots + a_i$, and this must be done in parallel in order to minimize execution time.

Kogge-Stone Scan The Kogge-Stone algorithm is a parallel inclusive scan algorithm that divides the computation into $\log n$ phases, where n is the size of the input array.

At each phase k, where k ranges from 0 to $\log n - 1$, each thread i adds the current value

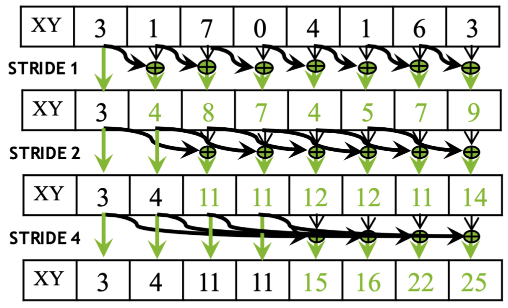


Figure 6: Example of execution of the Kogge-Stone prefix sum algorithm.

a_i to the value of the element that is at a distance of 2^k, if it exists. At the next phase, the ”range” of the sum, called stride, doubles. After the $\log n$ phases, each element contains the correct scan value.

In the execution of the algorithm, the total number of sums is:

$$(n-1) + (n-2) + (n-4) + \ldots + (n - n/2)$$
$$\approx n \log n - (n-1)$$

so the computational complexity is $\mathcal{O}(n \log n)$.

This algorithm is implemented in the `KoggeStoneScan` kernel, and the code is provided below:

```
__global__ void KoggeStoneScan(
  unsigned int *cdf,
  const unsigned int *histogram) {
  __shared__ unsigned int cache[NUM_BINS];
  const unsigned int tid = blockIdx.x *
      blockDim.x + threadIdx.x;

  if (tid < NUM_BINS) {
    cache[threadIdx.x] = histogram[tid];
  }

  for (unsigned int stride = 1; stride <
      blockDim.x; stride *= 2) {
    __syncthreads();
    unsigned int temp = 0;
    if (threadIdx.x >= stride) {
      temp = cache[threadIdx.x - stride];
    }
    __syncthreads();

    if (threadIdx.x >= stride) {
      cache[threadIdx.x] += temp;
    }
  }

  if (tid < NUM_BINS) {
    cdf[tid] = cache[threadIdx.x];
  }
}
```

- Each thread calculates its global index `tid`, which represents the position of the corresponding element in the array. Threads with `tid` smaller than NUM_BINS read the value from the histogram array and store it in the shared memory array `cache`.
- All threads in the block operate on the same shared memory area `cache`. Each thread participates in the cumulative scan, accumulating intermediate results.
- After each iteration, the number of active threads is halved.
- Finally, each thread is responsible for transferring one of the cumulative values calculated in the `cache` buffer to the `cdf` array, which is located in global memory and represents the cumulative distribution of the histogram.

Kogge-Stone Double Buffer Scan The Kogge-Stone Double Buffer algorithm builds upon the Kogge-Stone algorithm by modifying it to use two buffers in shared memory. Specifically, the two buffers are first initialized with the data, and then alternately, in each iteration/phase, one is used as input and the other as output. For example, assuming two buffers A and B:

- Phase 1: A as input and B as output
- Phase 2: B as input and A as output
- Phase 3: A as input and B as output
- ...

This approach reduces the overhead caused by synchronization between the threads and allows for greater parallelization. The kernel `KoggeStoneScanDoubleBuffer` implements this algorithm:

```
__global__ void KoggeStoneScanDoubleBuffer(
  unsigned int *cdf,
  const unsigned int *histogram) {
  __shared__ unsigned int cache[NUM_BINS];
  __shared__ unsigned int cacheAux[NUM_BINS];

  unsigned int *inputBuffer = cache;
  unsigned int *outputBuffer = cacheAux;

  const unsigned int tid = threadIdx.x +
      blockIdx.x * blockDim.x;
  if (tid < NUM_BINS) {
    cache[threadIdx.x] = histogram[tid];
  }

  for (unsigned int stride = 1; stride <
      blockDim.x; stride *= 2) {
    __syncthreads();

    if (threadIdx.x >= stride) {
      outputBuffer[threadIdx.x] =
          inputBuffer[threadIdx.x] +
              inputBuffer[threadIdx.x - stride];
    } else {
      outputBuffer[threadIdx.x] =
          inputBuffer[threadIdx.x];
    }

    unsigned int *temp = inputBuffer;
    inputBuffer = outputBuffer;
    outputBuffer = temp;
  }

  if (tid < NUM_BINS) {
    cdf[tid] = outputBuffer[threadIdx.x];
  }
}
```

This approach is implemented by using two pointers, `inputBuffer` and `outputBuffer`, which swap their contents between iterations. This way, the need for the second `__syncthreads()` call, which is required in the basic Kogge-Stone algorithm implementation, is eliminated.

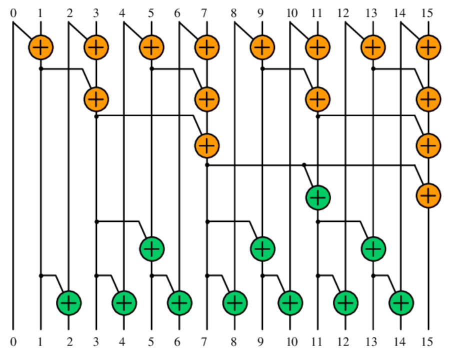


Figure 7: Example of execution of the Brent-Kung prefix sum algorithm.

Brent-Kung The Brent-Kung algorithm consists of two main phases:

- **Reduction Phase**: The data is combined hierarchically, progressively reducing the number of operations and creating a binary tree structure. In this phase, each node adds the value of another node at an increasing distance and propagates the result upwards.
- **Post Reduction Reverse Phase**: Starting from the root of the tree, the result is propagated downwards, updating the values so that each element contains the correct prefix sum.

In total, $2 \log n$ iterations are performed, and the total number of sums is the sum of the operations in the two phases, i.e.:

$$\begin{aligned} S_{red} &= \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots + 1 \\ &= \sum_{i=0}^{\log n - 1} \frac{n}{2^{i+1}} \\ &= n - 1 \end{aligned}$$

$$\begin{aligned} S_{postred} &= (2-1) + \ldots + \left(\frac{n}{4} - 1\right) + \left(\frac{n}{2} - 1\right) \\ &= \sum_{i=1}^{\log n - 1} \frac{n}{2^i} - \sum_{i=1}^{\log n - 1} 1 \\ &= (n-2) - (\log n - 1) \end{aligned}$$

Thus, the total number of sums is $S_{red} + S_{postred} = 2(n-1) - \log n$, and therefore the computational complexity is $\mathcal{O}(n)$.

It can also be observed that the total number of sums is no greater than twice that of the sequential prefix sum implementation, which involves exactly n sums.

The `BrentKungScan` kernel is implemented as follows:

```
__global__ void BrentKungScan(
  unsigned int *cdf,
  const unsigned int *histogram) {
  __shared__ unsigned int cache[NUM_BINS];
  const unsigned int tid = 2 * blockIdx.x *
      blockDim.x + threadIdx.x;
  if (tid < NUM_BINS) {
    cache[threadIdx.x] = histogram[tid];
  }

  if (tid + blockDim.x < NUM_BINS) {
    cache[threadIdx.x + blockDim.x] =
        histogram[tid + blockDim.x];
  }

  for (unsigned int stride = 1; stride <=
      blockDim.x; stride *= 2) {
    __syncthreads();

    unsigned int index = (threadIdx.x + 1) * 2 *
        stride - 1;
    if (index < NUM_BINS) {
      cache[index] += cache[index - stride];
    }
  }

  for (unsigned int stride = blockDim.x / 2;
      stride > 0; stride /= 2) {
    __syncthreads();

    unsigned int index = (threadIdx.x + 1) *
        stride * 2 - 1;
    if (index + stride < NUM_BINS) {
      cache[index + stride] += cache[index];
    }
  }

  __syncthreads();
  if (tid < NUM_BINS) {
    cdf[tid] = cache[threadIdx.x];
  }
  if (tid + blockDim.x < NUM_BINS) {
    cdf[tid + blockDim.x] = cache[threadIdx.x +
        blockDim.x];
  }
}
```

- Each thread calculates its global index `tid` and loads two values into shared memory: `histogram[tid]` and `histogram[tid + blockDim.x]`.
- The Reduction Phase is implemented through a for loop that initializes stride to 1, doubling it at each iteration to create a binary hierarchy. In each iteration, the specific index of the element to update is calculated as `(threadIdx.x + 1) * 2 * stride - 1`, and if it is within the limits of the array in shared memory, i.e., $2\times$BLOCK_SIZE, `cache[index]` is updated by adding `cache[index - stride]`, accumulating partial sums. The process repeats by doubling stride until it becomes greater than BLOCK_SIZE.
- A for loop is initialized that sets stride to

BLOCK_SIZE/2, halving it with each iteration, performing the Post Reduction Reverse Phase. In each iteration, the specific index of the element to update `index` is calculated as previously, and if `index - stride` is within the limits of the array in shared memory, `cache[index]` is updated by adding `cache[index + stride]`, propagating the results. The process repeats halving stride until it reaches 0.

- Each thread with `tid` smaller than NUM_BINS transfers `cache[threadIdx.x]` to `cdf[tid]`, and if `tid + BLOCK_SIZE` is also smaller than NUM_BINS it transfers `cache[threadIdx.x + blockDim.x]` to `cdf[tid + blockDim.x]`.

3. Benchmarking and Performance Analysis

The experiments were conducted on a machine with the following specification:

- CPU: Intel(R) Core(TM) i5-12400F
- GPU: NVIDIA RTX 3060 Ti
- RAM: 16 GB DIMM DDR4 3600 MHz CL17
- Storage: PCIe 4.0 NVMe SSD
- OS: Ubuntu 24.04.1 LTS

The NVIDIA RTX 3060 Ti GPU has 100 KB of shared memory per Streaming Multiprocessor (SM) and a maximum of 1024 threads per block. The block size was varied between 256 and 512 to evaluate the impact of thread granularity on execution time. Each implementation was tested on images of dimensions ranging from 1024×1024 to 8192×8192, representative of small to large computational workloads. For each implementation, the average execution time and the associated speedup are calculated by performing 10^3 executions.

All the images used were taken from the official European Space Agency (ESA) website dedicated to the Hubble Space Telescope[1], with all credits to NASA/ESA.

3.1. C++ Implementation

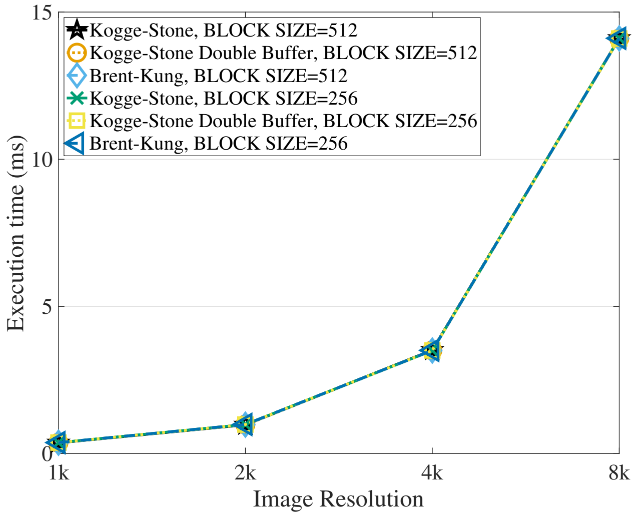


Figure 8: Graph showing the execution times recorded for varying image sizes in C++.

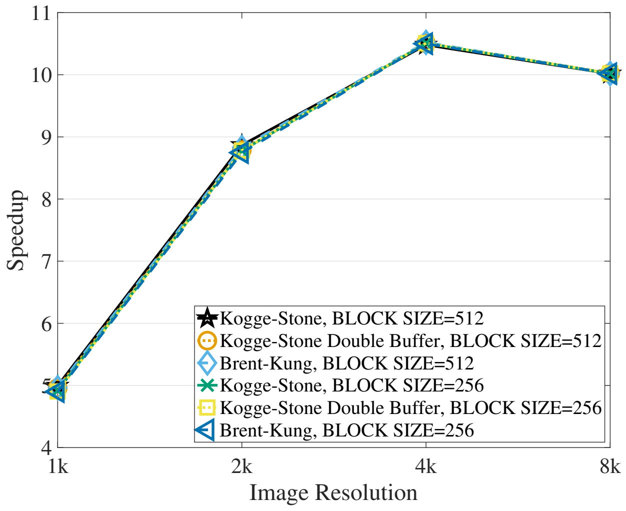


Figure 9: Graph showing the speedups recorded for varying image sizes in C++.

Table 1 summarizes the execution times for sequential and parallel implementations across different image sizes and block sizes. The execution times are graphically shown in Figure 8.

The results clearly highlight a significant improvement in execution times when moving from the sequential implementation to the parallel ones. For instance, for the 1024×1024 image, the execution time is 1.83429 ms, while the paral-

Image Resolution	**Sequential Implementation***	**Kogge-Stone***		**Kogge-Stone DB***		**Brent-Kung***	
		256	***512***	***256***	***512***	***256***	***512***
1024×1024	1.83429	0.372956	0.367343	0.374107	0.369951	0.374339	0.369508
2048×2048	8.67020	0.987788	0.977846	0.98731	0.986502	0.991279	0.982195
4096×4096	36.8264	3.50452	3.51354	3.50429	3.50495	3.50707	3.50033
8192×8192	141.328	14.1078	14.1086	14.1019	14.1004	14.1087	14.1048

*All times are expressed in milliseconds.

Table 1: Execution times for the various C++ implementations tested with varying image sizes and number of threads per block.

Image Resolution	**Kogge-Stone**		**Kogge-Stone DB**		**Brent-Kung**	
	256	***512***	***256***	***512***	***256***	***512***
1024×1024	4.91825	4.9934	4.90311	4.9582	4.90008	4.96414
2048×2048	8.77739	8.86663	8.78164	8.78883	8.74648	8.82737
4096×4096	10.5083	10.4813	10.5089	10.5070	10.5006	10.5208
8192×8192	10.0177	10.0172	10.0219	10.0230	10.0170	10.0199

Table 2: Speedup achieved by the various C++ implementations tested with varying image sizes and number of threads per block.

lel implementations (Kogge-Stone, Kogge-Stone Double Buffer, and Brent-Kung) achieve times around 0.37 ms. This improvement becomes even more pronounced with larger images, such as 8192×8192, where the parallel implementations reach times of about 14 ms compared to 141 ms for the sequential version.

When comparing the speedups, it can be observed that for 1024×1024 images, the speedup is around 4.9x, while for larger images, the speedup increases further, reaching values above 10x for 4096×4096 and 8192×8192 images. This behavior is consistent with the parallelism model, meaning that as the data size increases, the kernel launch overhead and memory management become negligible compared to the benefits of parallelism.

Looking at the results of the different scan variants, it is evident that Kogge-Stone Double Buffer is slightly faster than the basic Kogge-Stone version and shows similar performance to Brent-Kung. However, the performance differences between the variants are generally modest.

Regarding the number of threads per block, it can also be observed that the performance differences are not very significant, but generally the execution times of the parallel implementations achieved with 512 threads per block are lower than those with 256 threads per block.

3.2. Python Implementation

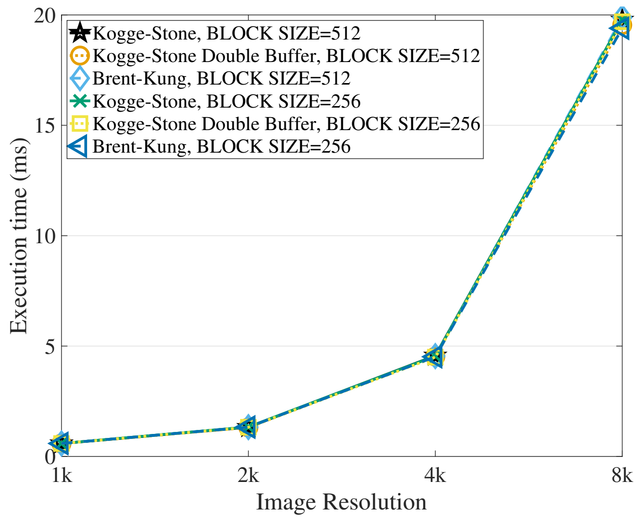


Figure 10: Graph showing the execution times recorded across different image sizes in Python.

Table 3 shows the execution times for the sequential and parallel implementations in Python using PyCUDA across different image sizes and block sizes. The speedups obtained for each implementation are reported in Table 4, while the

Image Resolution	Sequential Implementation*	Kogge-Stone*		Kogge-Stone DB*		Brent-Kung*	
		256	***512***	***256***	***512***	***256***	***512***
1024×1024	157.409	0.588307	0.586762	0.587753	0.585876	0.589077	0.588129
2048×2048	635.882	1.33628	1.32701	1.32488	1.3136	1.33119	1.32888
4096×4096	2515.29	4.55209	4.56166	4.50042	4.52382	4.51677	4.54483
8192×8192	10082.3	19.7829	19.7736	19.6921	19.5458	19.3966	19.8883

* All times are expressed in milliseconds.

Table 3: Execution times for the various Python implementations tested across different image sizes and block sizes.

Image Resolution	Kogge-Stone		Kogge-Stone DB		Brent-Kung	
	256	***512***	***256***	***512***	***256***	***512***
1024×1024	267.563	268.267	267.815	268.673	267.213	267.644
2048×2048	475.860	479.185	479.953	484.075	477.680	478.510
4096×4096	552.557	551.398	558.901	556.010	556.878	553.439
8192×8192	509.647	509.887	511.997	515.829	519.797	506.946

Table 4: Speedup achieved by the various Python implementations tested across different image sizes and block sizes.

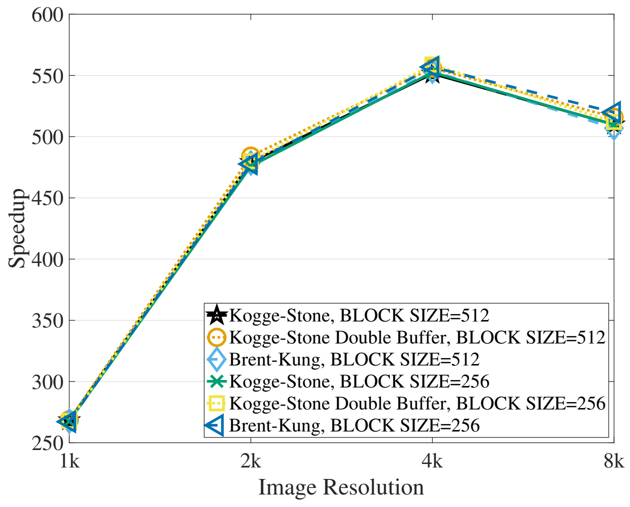


Figure 11: Graph showing the speedups recorded across different image sizes in Python.

execution times and speedups are visually represented in Figures 10 and 11, respectively.

The results highlight the significant advantage of parallel implementations compared to the sequential one, even in a Python environment, despite the inherent overhead introduced by the Python runtime. For instance, on a 1024×1024 image, the sequential implementation takes 157.409 ms, while the parallel implementations complete the task in approximately 0.59 ms, achieving a substantial speedup of around 268x. Similar trends are observed for larger images, where the parallel implementations outperform the sequential approach by a large margin. For example, on an 8192×8192 image, the sequential implementation takes 10082.3 ms, compared to about 19.7 ms for the parallel methods, resulting in a speedup of nearly 512x.

Comparing the three parallel scan algorithms, Kogge-Stone, Kogge-Stone Double Buffer, and Brent-Kung, the performance differences remain minimal. The Kogge-Stone Double Buffer implementation slightly outperforms the others in most cases, but the overall differences are negligible.

Regarding block size, as observed in the C++ implementation, the impact of thread granularity is not particularly pronounced in the Python experiments. The execution times and speedups are comparable between block sizes of 256 and 512 threads. However, in most cases, the configurations with 512 threads per block show marginally better performance.

These results further demonstrate that even when using Python, a higher-level language compared to C++, the efficiency of CUDA-based parallel implementations remains evident.

3.3. C++ vs Python

Table 5 and Figure 12 compare the execution times of the histogram equalization implemented in C++ and Python using the Kogge-Stone algo-

Image Resolution	**Kogge-Stone**	
	C++	***Python***
1024×1024	0.367343	0.586762
2048×2048	0.977846	1.32701
4096×4096	3.51354	4.56166
8192×8192	14.1086	19.7736

Table 5: Execution times of histogram equalization implemented in C++ and Python using Kogge-Stone as the scan algorithm and 512 threads per block.

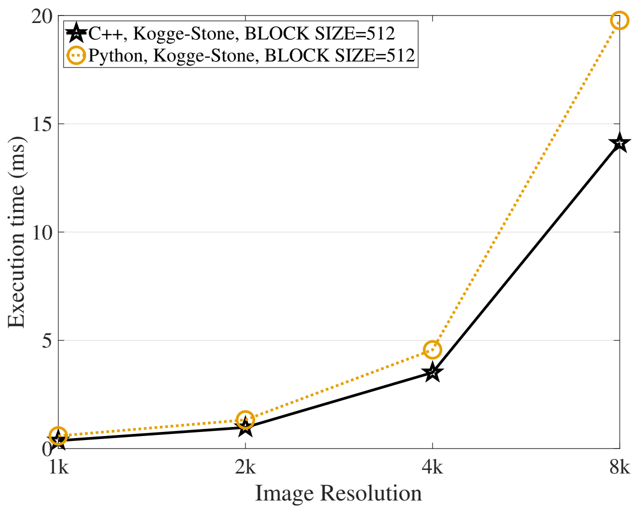


Figure 12: Graph showing the execution time of the parallel implementation in C++ and Python using Kogge-Stone across different image sizes, with a block size of 512.

rithm with 512 threads per block. The results highlight the expected performance gap between the two languages, with the C++ implementation consistently outperforming the Python implementation across all image sizes.

For instance, on an image of size 1024×1024, the C++ implementation achieves an execution time of 0.367343 ms, while the Python implementation takes 0.586762 ms, resulting in a roughly 1.6x difference. Similarly, for the largest image size tested, 8192×8192, the C++ implementation completes in 14.1086 ms, whereas the Python version requires 19.7736 ms, maintaining a similar performance ratio.

The performance discrepancy can be attributed to several factors, including:

1. **Runtime Overhead**: Python, being a high-level interpreted language, incurs additional runtime overhead compared to C++, a compiled language. This overhead arises from Python's dynamic typing, garbage collection, and the abstraction layers introduced by PyCUDA

2. **Kernel Launch Overhead in PyCUDA**: although PyCUDA provides a user-friendly interface for CUDA programming, it introduces additional overhead when launching kernels compared to a native CUDA implementation in C++

3. **Compiler Optimization**: C++ code benefits from advanced compiler optimizations during the build process, such as loop unrolling, inlining, and register allocation. These optimizations help reduce execution time significantly

Despite these differences, it is worth noting that the Python implementation still delivers strong performance and remains competitive, achieving execution times within 1.2x to 1.6x of the C++ implementation. This demonstrates that Python, with PyCUDA, is a viable option for GPU-accelerated computing, especially for applications where development speed and simplicity are prioritized over maximum performance.

4. Conclusions

The experimental results highlight the significant advantages of parallel histogram equalization over its sequential counterpart, with substantial performance improvements observed across all tested implementations. The parallel approaches, based on Kogge-Stone, Kogge-Stone Double Buffer, and Brent-Kung algorithms, demonstrate increasing efficiency as image size grows, confirming the scalability of GPU-based solutions for this task.

The comparison between C++ and Python implementations reveals a consistent performance gap, with the C++ version achieving execution times up to 1.6× faster due to reduced runtime overhead, more efficient kernel execution, and compiler optimizations. Nonetheless, the Python implementation using PyCUDA remains a viable alternative, offering competitive performance.

Overall, the results confirm the high effectiveness of GPU-based parallel histogram equalization.

References

[1] NASA/ESA. SM3A: Graceful Hubble. `https://esahubble.org/images/sts103_726_081/`. 9

[2] W. Hwu, D. Kirk, and I. Hajj. Programming Massively Parallel Processors: A Hands-on Approach, Fourth Edition. Elsevier, Jan. 2022. Publisher Copyright: © 2023 Elsevier Inc. All rights reserved. 5

[3] D. Kirk and W. mei W. Hwu. Parallel computation patterns – parallel scan (prefix sum). `https://lumetta.web.engr.illinois.edu/408-S20/slide-copies/ece408-lecture16-S20.pdf`, 2020. Lecture slides for ECE408 Spring 2020, University of Illinois.

[4] Wikipedia. Histogram equalization. `https://en.wikipedia.org/wiki/Histogram_equalization`.