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: Esempio concreto di applicazione dell'histogram equalization. A sinistra l'immagine originale, a destra l'immagine equalizzata.

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.

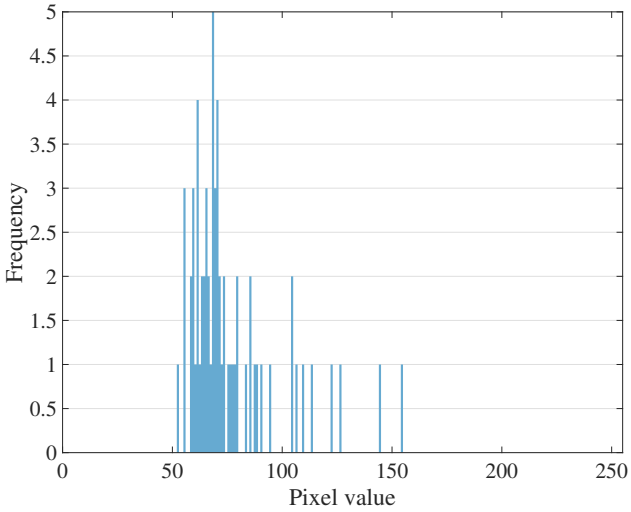

Figure 3: Histogram of the 8x8 grayscale image.

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

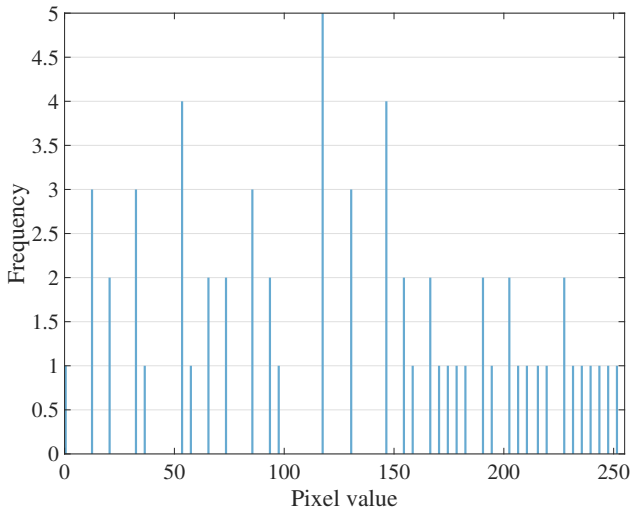

Figure 4: Histogram of the equalized 8x8 grayscale image.

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

Initialize $hist \leftarrow$ array of size L filled with zeros
for each pixel p in $image$ **do**
    $hist[p] \leftarrow hist[p] + 1$
end for
Initialize $cdf \leftarrow$ array of size L filled with zeros
$cdf[0] \leftarrow hist[0]$
for $i = 1$ to $L-1$ **do**
    $cdf[i] \leftarrow cdf[i-1] + hist[i]$
end for
$cdf\_min \leftarrow$ first nonzero element in cdf
$total\_pixels \leftarrow M \times N$
for $i = 0$ to $L-1$ **do**
    $cdf[i] \leftarrow \text{round}\left(\frac{cdf[i]-cdf\_min}{total\_pixels-cdf\_min} \times (L-1)\right)$
end for
for each pixel p in $image$ **do**
    $equalized\_image[p] \leftarrow cdf[image[p]]$
end for
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 = static_cast<unsigned int>(
        round(static_cast<double>(value -
            cdfMin) / (pixelCount - cdfMin) *
              (NUM_BINS - 1)));
  }

  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];
  if (threadIdx.x < NUM_BINS) {
    cache[threadIdx.x] = 0;
  }

  __syncthreads();

  if (const unsigned int tid = threadIdx.x +
      blockDim.x * blockIdx.x;
      tid < pixelCount) {
    atomicAdd(&(cache[input[tid]]), 1);
  }

  __syncthreads();

  if (threadIdx.x < NUM_BINS) {
    atomicAdd(&(histogram[threadIdx.x]),
        cache[threadIdx.x]);
  }
}
```

This is executed with a 1D grid composed of a number of blocks equal to:

$$\left\lceil \frac{\text{num. pixel}}{\text{num. threads per blocco}} \right\rceil \tag{2}$$

in order to cover all the pixels and process them.

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. Then, each thread calculates its

global index and updates the histogram using the `atomicAdd` function, which guarantees atomic operations to avoid race conditions. The second `__syncthreads()` ensures that all threads have completed the `cache` update. Finally, threads with indices relative to their block that are smaller than the number of bins (256) add the values calculated in the `cache` to the global histogram using another `atomicAdd` call.

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) {
  if (const unsigned int tid = threadIdx.x +
      blockIdx.x * blockDim.x;
      tid < pixelCount) {
    output[tid] = cdf[input[tid]];
  }
}
```

The mapping of the image to the normalized CDF is executed with a 1D grid composed of a number of blocks as given in Equation 2.

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.

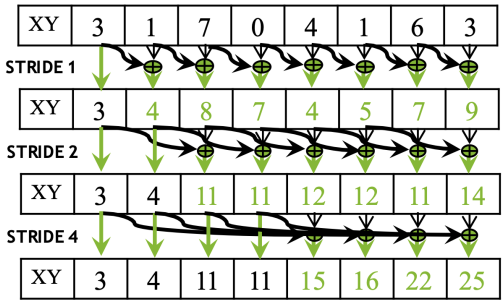

Figure 6: Example of execution of the Kogge-Stone prefix sum algorithm.

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

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.

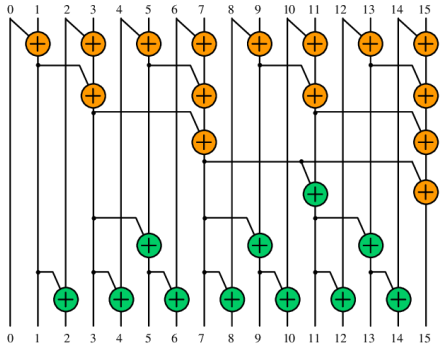

Figure 7: Example of execution of the Brent-Kung prefix sum algorithm.

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×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 512 and 1024 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

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 parallel implementations (Kogge-Stone, Kogge-Stone

Image Resolution	Sequential Implementation*	Kogge-Stone*		Kogge-Stone DB*		Brent-Kung*	
		512	*1024*	*512*	*1024*	*512*	*1024*
1024×1024	1.83429	0.390898	0.388011	0.391095	0.388066	0.390925	0.389134
2048×2048	8.67020	1.05540	1.05017	1.05838	1.05216	1.05537	1.05144
4096×4096	36.8264	3.76853	3.74518	3.77771	3.73870	3.76234	3.74462
8192×8192	141.328	15.1101	15.1372	15.1756	15.0845	15.1068	15.0358

*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	
	512	*1024*	*512*	*1024*	*512*	*1024*
1024×1024	4.63279	4.66726	4.63046	4.66660	4.63247	4.65379
2048×2048	7.69899	7.73730	7.67726	7.72272	7.69919	7.72801
4096×4096	8.88000	8.93537	8.85844	8.95085	8.89464	8.93671
8192×8192	8.87189	8.85602	8.83358	8.88694	8.87383	8.91570

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

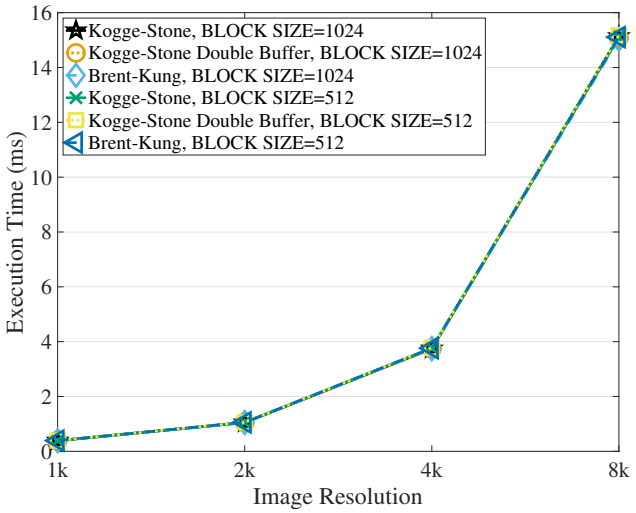

Figure 8: Graph showing the execution times recorded for varying image sizes in C++.

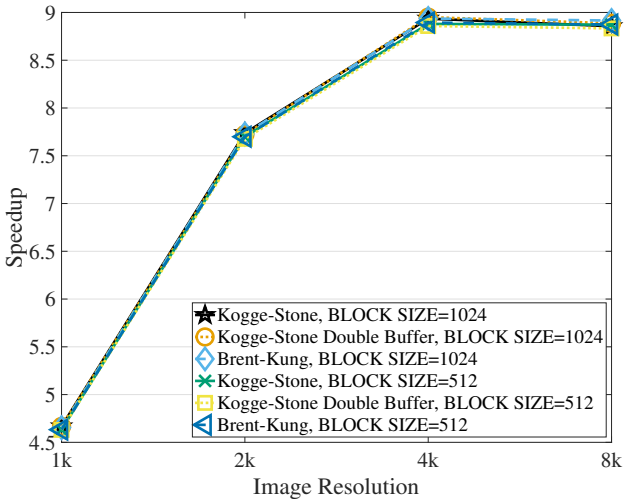

Figure 9: Graph showing the speedups recorded for varying image sizes in C++.

Double Buffer, and Brent-Kung) achieve times around 0.39 ms. This improvement becomes even more pronounced with larger images, such as 8192×8192, where the parallel implementations reach times of about 15 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 4.6-4.7x, while for larger images, the speedup increases further, reaching values above 8.8x 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 1024 threads per block are lower than those with 512 threads per block.

3.2. Python Implementation

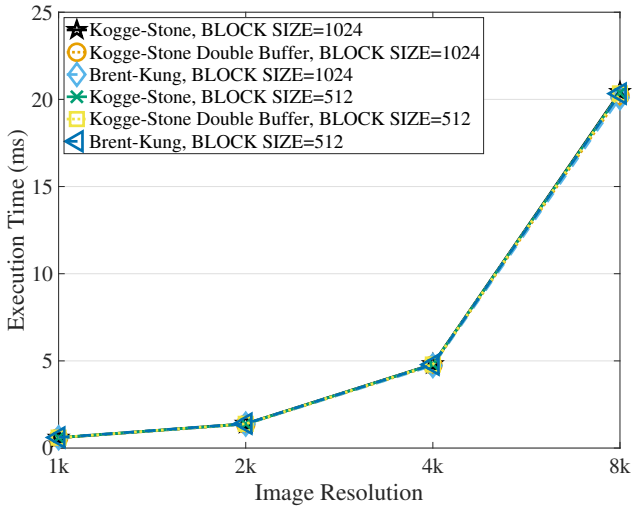

Figure 10: Graph showing the execution times recorded across different image sizes in Python.

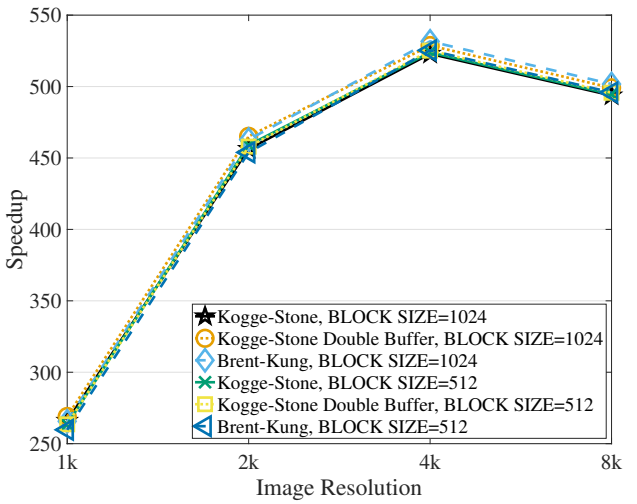

Figure 11: Graph showing the speedups recorded across different image sizes in Python.

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 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 0.59-0.60 ms, achieving a substantial speedup of around 263x. 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 20.3 ms for the parallel methods, resulting in a speedup of nearly 495x.

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 some 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 512 and 1024 threads. However, in most cases, the configurations with 1024 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 algorithm with 1024 threads per block. The results

Image Resolution	Sequential Implementation*	Kogge-Stone*		Kogge-Stone DB*		Brent-Kung*	
		512	*1024*	*512*	*1024*	*512*	*1024*
1024×1024	157.409	0.597348	0.591853	0.597255	0.58522	0.605941	0.592609
2048×2048	635.882	1.38548	1.39363	1.38861	1.36675	1.40058	1.37374
4096×4096	2515.29	4.80036	4.8104	4.79013	4.7581	4.78619	4.73131
8192×8192	10082.3	20.3792	20.4211	20.3427	20.2019	20.33048	20.0954

* 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	
	512	*1024*	*512*	*1024*	*512*	*1024*
1024×1024	263.520	266.003	263.554	269.014	259.720	265.654
2048×2048	459.022	456.343	457,927	465.151	453.895	462.856
4096×4096	523.893	522.941	525.170	528.697	525.355	531.645
8192×8192	494.536	493.751	495.669	499.103	495.869	501.751

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

Image Resolution	Kogge-Stone	
	C++	*Python*
1024×1024	0.388011	0.591853
2048×2048	1.05017	1.39363
4096×4096	3.74518	4.8104
8192×8192	15.1372	20.4211

Table 5: Execution times of histogram equalization implemented in C++ and Python using Kogge-Stone as the scan algorithm and 1024 threads per block.

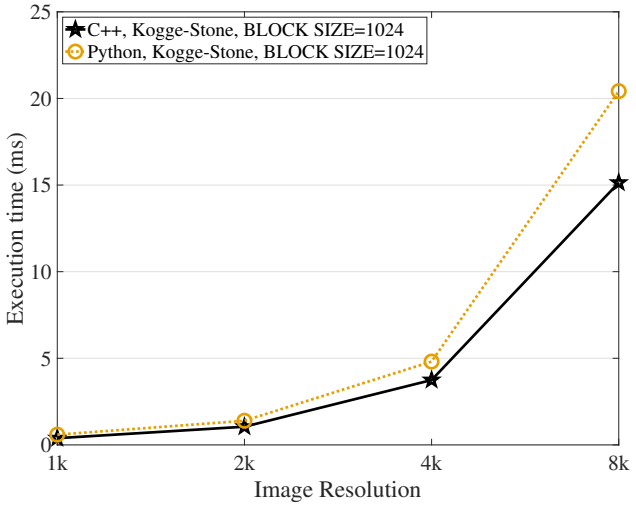

Figure 12: Graph showing the execution time of the parallel implementation in C++ and Python using Kogge-Stone across different image sizes, with a block size of 1024.

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 0.388011 ms, while the Python implementation takes 0.591853 ms, resulting in a roughly 1.5x difference. Similarly, for the largest image size tested (8192×8192), the C++ implementation completes in 15.1372 ms, whereas the Python version requires 20.4211 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 1.3x to 1.5x 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 1.5× 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/. 8

[2] 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.

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