PC Lab-2024/25 Integral Image

Alessio Bugetti
alessio.bugetti@edu.unifi.it

Abstract

This report presents a comprehensive performance evaluation of integral image computation on grayscale images, comparing sequential and parallel implementations across multiple platforms. The sequential version is implemented in C++, while the parallel implementations are developed using native CUDA C++, PyCUDA, and Numba. To assess performance, the system is tested with image sizes ranging from 1K to 8K, and the average execution time is measured over 1,000 runs for each configuration. In the parallel implementations, key GPU parameters such as SECTION_SIZE and BLOCK_ROWS are varied to analyze their impact on performance. The speedup of the parallel versions is computed relative to the sequential implementation. A detailed comparison of execution times across native CUDA C++, PyCUDA, and Numba is also provided.

1. Introduction

The goal of this project is to implement integral image processing both sequentially on the CPU and in parallel on the GPU using CUDA, while analyzing the resulting speedup. Integral images are a fundamental technique in image processing and computer vision, widely used to accelerate computations involving summations over rectangular regions.

This method is particularly beneficial for applications such as object detection, texture analysis, and feature extraction, where repeated summation of pixel intensities is required.

By precomputing an integral image representation, these summations can be performed in constant time, significantly improving computational efficiency.

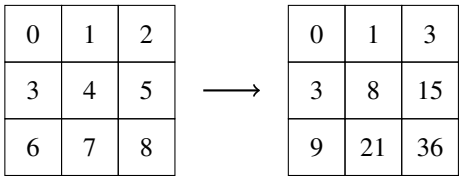

Figure 1: Integral image of a 3x3 example image.

1.1. Integral Image Computation

The value of the integral image at a point (x, y) is given by the sum of all points in the rectangle from the origin to (x, y):

$$I(x,y) = \sum_{\substack{x' \leq x \\ y' \leq y}} i(x', y')$$

where $i(x, y)$ represents the intensity of the original image at (x, y). The integral image can be efficiently computed in a single pass since its value can be rewritten as:

$$I(x,y) = i(x,y) + I(x,y-1) + I(x-1,y) - I(x-1,y-1)$$

Taking as an example the 3x3 image shown in Figure 1, which illustrates the construction of the integral image, it can be observed that for the pixel at $(1, 2)$:

$$\begin{aligned} I(1,2) &= i(1,2) + I(1,1) + I(0,2) - I(0,1) \\ &= 5 + 8 + 3 - 1 \\ &= 15 \end{aligned}$$

2. Project Structure

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

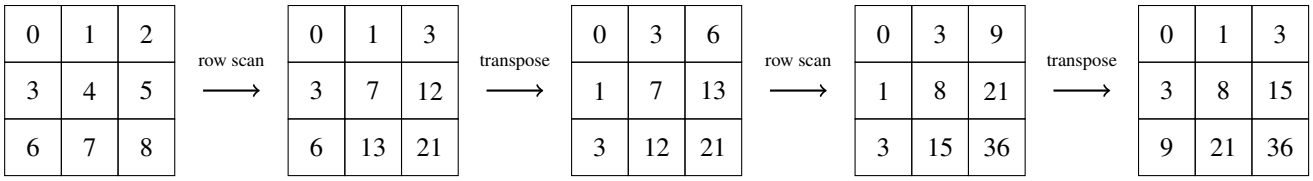

Figure 2: Example execution of the algorithm that computes the integral image implemented in the parallel version with CUDA.

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

```
./integralimage build
```

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

```
./integralimage run
```

Instead, to run the benchmarks for the PyCUDA implementation:

```
python python/pycuda_test.py
```

And finally, to run the benchmarks for the Numba implementation:

```
python python/numba_test.py
```

2.1. Sequential Implementation

Below is the pseudocode for the sequential implementation:

The algorithm follows these steps:

- A matrix B of the same dimensions as the original image A is allocated
- The initial cell $B[1][1]$ is set equal to the initial value of $A[1][1]$
- The first column is filled by cumulatively summing the values of the corresponding column in A
- Similarly, the first row is populated by summing the values of the corresponding row in A

Algorithm 1 Sequential Integral Image

Require: A: input image of size $N \times M$
Ensure: B: output integral image of the same size

$B[1][1] \leftarrow A[1][1]$
for $i = 2$ to N **do**
  $B[i][1] \leftarrow B[i-1][1] + A[i][1]$
end for
for $j = 2$ to M **do**
  $B[1][j] \leftarrow B[1][j-1] + A[1][j]$
end for
for $i = 2$ to N **do**
  for $j = 2$ to M **do**
    $B[i][j] \leftarrow A[i][j] + B[i][j-1] + B[i-1][j] - B[i-1][j-1]$
  end for
end for
return B

- For every other cell $B[i][j]$, the value is computed as the sum of the original image value at that position, plus the cumulative values from the previous cells in the same row and the same column, minus the overlap already accounted for in $B[i-1][j-1]$

This approach ensures an efficient computation of the Integral Image in $\mathcal{O}(NM)$ time.

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

```
double
SequentialIntegralImage(
  const unsigned int * input,
  unsigned int * output,
  const unsigned int height,
  const unsigned int width) {
  const auto start =
      std::chrono::high_resolution_clock::now();

  output[0] = input[0];

  for (unsigned int i = 1; i < height; i++) {
    output[i * width] = input[i * width] +
        output[(i - 1) * width];
  }
```

```
  for (unsigned int j = 1; j < width; j++) {
    output[j] = input[j] + output[j - 1];
  }

  for (unsigned int i = 1; i < height; i++) {
    for (unsigned int j = 1; j < width; j++) {
      output[i * width + j] = input[i * width +
          j] + output[(i - 1) * width + j] +
        output[i * width + j - 1] - output[(i -
            1) * width + j - 1];
    }
  }
  const auto stop =
      std::chrono::high_resolution_clock::now();

  return std::chrono::duration < double,
      std::milli > (stop - start).count();
}
```

2.2. Parallel Implementation with CUDA

Regarding the parallel implementation with CUDA, l'algoritmo proposto, visibile in Figure 2, prevede l'alternanza di due operazioni, ovvero row-wise inclusive scan e traposizione della matrice, nel seguente ordine:

- Row-wise Scan
- Transpose
- Row-wise Scan
- Transpose

Per quanto riguarda il row-wise inclusive scan, sono stati definiti due kernel diversi che la realizzano. Nella prima versione, che è quella naive, ogni thread è responsabile di una specifica riga della matrice e semplicemente scorrendo la riga da sinistra a destra, somma ogni elemento con il precedente:

Regarding the parallel implementation with CUDA, the proposed algorithm, shown in Figure 2, alternates between two operations: row-wise inclusive scan and matrix transposition, in the following order:

1. Row-wise Scan
2. Transpose
3. Row-wise Scan
4. Transpose

For the row-wise inclusive scan, two different kernels have been implemented. In the first version, which is the naive approach, each thread is responsible for a specific row of the matrix and, by simply scanning the row from left to right, sums each element with the previous one:

```
__global__ void
SumRows(
  unsigned int * input,
  const unsigned int height,
  const unsigned int width) {
  const unsigned int tid = blockIdx.y *
      blockDim.y + threadIdx.y;

  if (tid < height) {
    for (int j = 1; j < width; j++) {
      input[tid * width + j] += input[tid *
          width + (j - 1)];
    }
  }
}
```

In the other version, the optimized approach, the Kogge-Stone algorithm is used for the scan.

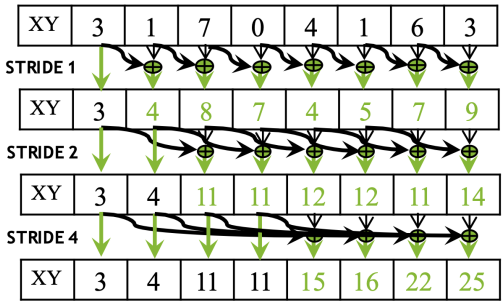

Figure 3: Example of execution of the Kogge-Stone prefix sum algorithm.

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

In each phase k, where k ranges from 0 to $\log n - 1$, each thread i adds its current value a_i to the value of the element at a distance of 2^k, if it exists. In the next phase, the range of the sum, referred to as stride, doubles. After $\log n$ phases, each element contains the correct scan value.

During the execution of the algorithm, the total number of additions is:

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

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

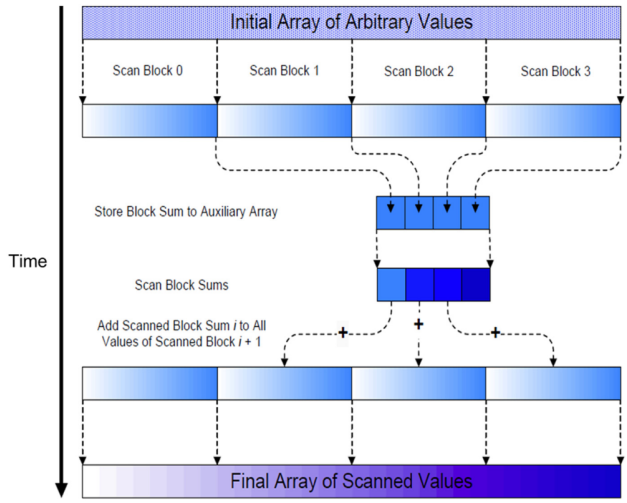

Figure 4: A hierarchical scan for arbitrary length inputs[1].

To handle cases where the maximum number of threads per block exceeds the number of elements in a row, a row-wise segmented scan is implemented. A traditional segmented scan divides the input into fixed-size blocks, each processed independently by a thread block. Each block performs a local scan on its own data. After this initial phase, partial results are written to global memory. Subsequently, a scan is executed only on the last value of each block, followed by a third kernel that updates the local results of the blocks with corrections derived from the previous phase. This approach, visually represented in Figure 4, introduces overhead due to frequent accesses to global memory between different phases.

The `SinglePassRowWiseScan` kernel implements the Kogge-Stone algorithm to perform the scan. However, instead of using a segmented approach, it manages the entire execution in a single pass. This reduces the number of accesses to global memory, thereby improving efficiency compared to a basic segmented version. The implementation is as follows:

```
__global__ void
SinglePassRowWiseScan(
  const unsigned int * input,
  unsigned int * output,
  unsigned int * flags,
  unsigned int * scanValue,
  unsigned int * blockCounter,
  const unsigned int numRows,
  const unsigned int numCols) {
  __shared__ unsigned int XY[SECTION_SIZE];
  __shared__ unsigned int bid_s;

  if (threadIdx.x == 0) {
    bid_s = atomicAdd(blockCounter, 1);
  }
  __syncthreads();

  const unsigned int bid = bid_s;
  const unsigned int blockIdx_x = bid / numRows;
  const unsigned int blockIdx_y = bid % numRows;
  const int col = blockIdx_x * SECTION_SIZE +
      threadIdx.x;
  const unsigned int row = blockIdx_y;

  const int pixel = row * numCols + col;

  if (row < numRows && col < numCols) {
    XY[threadIdx.x] = input[pixel];
  } else {
    XY[threadIdx.x] = 0;
  }

  for (int stride = 1; stride < SECTION_SIZE;
      stride *= 2) {
    __syncthreads();
    unsigned int tmp = 0;
    if (threadIdx.x >= stride) {
      tmp = XY[threadIdx.x - stride];
    }
    __syncthreads();
    if (threadIdx.x >= stride) {
      XY[threadIdx.x] += tmp;
    }
  }
  __syncthreads();
  __shared__ unsigned int previousSum;
  if (threadIdx.x == 0) {
    while (blockIdx_x >= 1 && atomicAdd( &
        flags[bid], 0) == 0) {}
    previousSum = scanValue[bid];
    scanValue[bid + numRows] = XY[blockDim.x -
        1] + previousSum;
    __threadfence();
    atomicAdd( & flags[bid + numRows], 1);
  }
  __syncthreads();

  if (row < numRows && col < numCols) {
    output[pixel] = XY[threadIdx.x] +
        previousSum;
  }
}
```

It can be observed that an atomic counter, `blockCounter`, is used to assign a logical identifier to each block, allowing the decoupling of the physical block index, managed by the system, from its logical identifier, managed by the kernel itself. This resolves the potential deadlock issue that could occur when blocks are executed

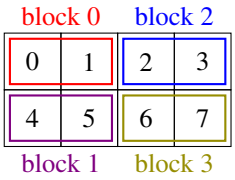

Figure 5: An example of the dynamic assignment technique for block indices and its effect on the matrix elements processed by each block is presented for a 2×4 image, where SECTION_SIZE is set to 2.

out of order with respect to the expected logical indices.

Consider an image with N rows and M columns where a dynamic assignment technique for block indices is not used. Suppose the GPU scheduler executes the blocks in the following order: $i, i+1, \ldots, i+K$, while blocks $i-N, i+1-N, \ldots, i+K-N$ have not yet been executed. This could lead to a complete deadlock, as the following situation might occur:

- Block i waits for the value from block $i-N$, and similarly, all other blocks $i+1, \ldots, i+K$ are waiting for values from blocks $i+1-N, \ldots, i+K-N$, respectively.
- If all multiprocessors are occupied by blocks i to $i+K$, block $i-N$ cannot be started.
- None of the already started blocks can complete their execution because they are all waiting for values from their predecessors.

This highlights the importance of using a dynamic assignment technique for block indices.

Specifically, each block is associated with a single row and processes a segment of it, following the logic illustrated in Figure 5.

The kernel operates using a sequential pass model where:

- Block 0 computes the partial sum of its data and passes it to block N.
- Block 1 computes the partial sum of its data and passes it to block $N+1$.
- ...
- Block $N-1$ computes the partial sum of its data and passes it to block $2N-1$.
- Block N adds the value received from block 0 to its local result and then passes it to block $2N$.
- This process continues until all blocks have completed their tasks and the cumulative values have been propagated throughout the grid.

Each block computes the partial sum of its data within its own shared space, `XY`. The threads within each block perform the scan operation in parallel using the Kogge-Stone algorithm. Each block executes this computation independently before waiting for results from the preceding blocks.

A key characteristic of this approach is *adjacent synchronization*, which allows thread blocks to synchronize and exchange data. This is implemented using a flag system to ensure that data is correctly transferred from one block to another. It can be observed that this phase is managed by a leader thread for each block, while the other threads wait using `__syncthreads()`. The leader thread continuously checks the `flags[bid]` flag until it is set. Once set, it reads `scanValue[bid]` from global memory, which contains the partial sum of the block identified as `bid-N`, and stores the value in the shared variable `previous_sum`. Subsequently, it adds `previous_sum` to `XY[blockDim.x - 1]`, which holds its partial sum, and stores the result in the `scanValue[bid+N]` array in global memory. The use of `__threadfence()` is crucial, as it ensures that the `scanValue[bid+N]` value has been written to global memory before setting `flag[bid+N]` using `atomicAdd()`.

The `Transpose` kernel, on the other hand, performs the transposition of a matrix:

```
__global__ void
Transpose(
  const unsigned int * input,
  unsigned int * output,
  const unsigned int height,
  const unsigned int width) {
```

```
  __shared__ unsigned int
      tile[TILE_DIM][TILE_DIM + 1];

  unsigned int x = blockIdx.x * TILE_DIM +
      threadIdx.x;
  unsigned int y = blockIdx.y * TILE_DIM +
      threadIdx.y;

  for (unsigned int j = 0; j < TILE_DIM; j +=
      BLOCK_ROWS) {
    if ((x < width) && (y + j < height)) {
      tile[threadIdx.y + j][threadIdx.x] =
          input[(y + j) * width + x];
    }
  }
  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (unsigned int j = 0; j < TILE_DIM; j +=
      BLOCK_ROWS) {
    if ((x < height) && (y + j < width)) {
      output[(y + j) * height + x] =
          tile[threadIdx.x][threadIdx.y + j];
    }
  }
}
```

This kernel, proposed by NVIDIA[4], was modified to work with non-square matrices as well. It uses a shared memory matrix of dimensions `[TILE_DIM, TILE_DIM+1]` where the `+1` is used to reduce the probability of bank conflicts.

During the first loop phase, a warp of threads executes a coalesced memory access pattern, reading consecutive elements from `input` and storing them into the rows of a shared memory tile. After an index remapping step, a column of the shared memory tile is then written to a contiguous memory region in `output`. As the data written by threads to `output` differs from the data they initially read from `input`, a block-wide barrier synchronization, achieved through `__syncthreads()`, is required.

The thread coarsening technique is also implemented, which means a loop is used to load multiple elements into shared memory per thread. Specifically, `BLOCK_ROWS` determines how many rows are processed by each thread.

2.3. PyCUDA Implementation

PyCUDA is a library that provides a Python interface for NVIDIA's CUDA API, enabling the utilization of GPU computing power directly from the Python language. One of PyCUDA's main features is the ability to integrate CUDA C++ code within a Python program, allowing for writing, compiling, and launching CUDA kernels on the GPU.

After writing the kernels, PyCUDA's `SourceModule` is used to compile the code. The `SourceModule` class accepts a string of CUDA code, compiles it, and makes it executable on the GPU. Once the code is compiled, CUDA functions can be obtained from the module using the `get_function()` method.

The usage for the `SinglePassRowWiseScan` and `Transpose` kernels is as follows:

```
try:
    with open("../c++/kernel.cu", "r") as f:
        kernel_code = f.read()
except FileNotFoundError:
    print("Error: kernel.cu file not found")
    sys.exit(1)

mod = SourceModule(kernel_code)
SinglePassRowWiseScan =
    mod.get_function("SinglePassRowWiseScan")
Transpose = mod.get_function("Transpose")
```

2.4. Numba Implementation

Numba is a Python library that enables code acceleration through Just-In-Time (JIT) compilation and supports execution on both CPU and GPU.

With Numba, which provides the `@cuda.jit` decorator, it is possible to rewrite kernels defined in Native CUDA C++ directly in Python while maintaining the same logic.

The `SinglePassRowWiseScan` kernel was rewritten as follows:

```
@cuda.jit
def single_pass_row_wise_scan(
        input, output, flags, scan_value,
            block_counter, num_rows, num_cols
):
    XY = cuda.shared.array(SECTION_SIZE,
        dtype=types.uint32)
    bid_s = cuda.shared.array(1,
        dtype=types.uint32)

    if cuda.threadIdx.x == 0:
        bid_s[0] =
            cuda.atomic.add(block_counter, 0, 1)
    cuda.syncthreads()
```

```
    bid = bid_s[0]
    blockIdx_x = bid // num_rows
    blockIdx_y = bid % num_rows
    col = blockIdx_x * SECTION_SIZE +
        cuda.threadIdx.x
    row = blockIdx_y

    pixel = row * num_cols + col

    if row < num_rows and col < num_cols:
        XY[cuda.threadIdx.x] = input[pixel]
    else:
        XY[cuda.threadIdx.x] = 0

    stride = 1
    while stride < SECTION_SIZE:
        cuda.syncthreads()
        tmp = 0
        if cuda.threadIdx.x >= stride:
            tmp = XY[cuda.threadIdx.x - stride]
        cuda.syncthreads()
        if cuda.threadIdx.x >= stride:
            XY[cuda.threadIdx.x] += tmp
        stride *= 2

    previous_sum = cuda.shared.array(1,
        dtype=types.uint32)
    if cuda.threadIdx.x == 0:
        while blockIdx_x >= 1 and
            cuda.atomic.add(flags, bid, 0) == 0:
            pass
        previous_sum[0] = scan_value[bid]
        scan_value[bid + num_rows] =
            XY[cuda.blockDim.x - 1] +
            previous_sum[0]
        cuda.threadfence()
        cuda.atomic.add(flags, bid + num_rows, 1)
    cuda.syncthreads()

    if row < num_rows and col < num_cols:
        output[pixel] = XY[cuda.threadIdx.x] +
            previous_sum[0]
```

Meanwhile, the `Transpose` kernel is implemented as follows:

```
@cuda.jit
def transpose(input, output, height, width):
    tile = cuda.shared.array((TILE_DIM,
        TILE_DIM_2), dtype=types.uint32)

    x = cuda.blockIdx.x * TILE_DIM +
        cuda.threadIdx.x
    y = cuda.blockIdx.y * TILE_DIM +
        cuda.threadIdx.y

    for j in range(0, TILE_DIM, BLOCK_ROWS):
        if x < width and (y + j) < height:
            tile[cuda.threadIdx.y +
                j][cuda.threadIdx.x] = input[(y
                + j) * width + x]

    cuda.syncthreads()

    x = cuda.blockIdx.y * TILE_DIM +
        cuda.threadIdx.x
    y = cuda.blockIdx.x * TILE_DIM +
        cuda.threadIdx.y

    for j in range(0, TILE_DIM, BLOCK_ROWS):
        if x < height and (y + j) < width:
            output[(y + j) * height + x] =
                tile[cuda.threadIdx.x]
                    [cuda.threadIdx.y + j]
```

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 SECTION_SIZE was varied between 512 and 1024 and BLOCK_ROWS between 8 and 16. Each configuration 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.

3.1. Native CUDA C++ Benchmarks

The results of the naive implementation of Integral Image Processing in CUDA C++ are presented in Table 1, which shows the execution times for different image resolutions and parameter combinations. As expected, the execution time increases with the image size, ranging from approximately 0.85 ms for 1024×1024 images to over 17 ms for 8192×8192 images. Additionally, the parameters SECTION_SIZE and BLOCK_ROWS significantly affect performance, with some configurations proving to be more efficient.

Image Resolution	Sequential Implementation*	SECTION_SIZE,(TILE_SIZE, BLOCK_ROWS)*			
		256,(32,8)	*256,(32,16)*	*512,(32,8)*	*512,(32,16)*
1024×1024	2.80113	0.851717	0.936136	1.65531	1.66337
2048×2048	11.4013	1.75311	1.94186	3.37271	3.73937
4096×4096	44.5048	3.84416	4.23549	7.09557	7.87032
8192×8192	175.950	12.1023	13.2162	15.5063	17.0288

*All times are expressed in milliseconds.

Table 1: Execution times for the CUDA C++ naive implementation tested with different image sizes, section sizes and block rows.

Image Resolution	SECTION_SIZE,(TILE_SIZE, BLOCK_ROWS)			
	256,(32,8)	*256,(32,16)*	*512,(32,8)*	*512,(32,16)*
1024×1024	3.2888	2.99223	1.69221	1.68401
2048×2048	6.50348	5.87133	3.38045	3.04899
4096×4096	11.5772	10.5076	6.2722	5.65476
8192×8192	14.5386	13.3132	11.3470	10.3325

Table 2: Speedup achieved by the CUDA C++ naive implementation tested with different image sizes, section sizes and block rows.

Image Resolution	Sequential Implementation*	SECTION_SIZE,(TILE_SIZE, BLOCK_ROWS)*			
		256,(32,8)	*256,(32,16)*	*512,(32,8)*	*512,(32,16)*
1024×1024	2.80113	0.135548	0.134093	0.14986	0.151053
2048×2048	11.4013	0.524591	0.528902	0.57176	0.57477
4096×4096	44.5048	2.1066	2.11648	2.29449	2.30238
8192×8192	175.950	8.37909	8.38083	9.12717	9.15439

*All times are expressed in milliseconds.

Table 3: Execution times for the CUDA C++ optimized implementation tested with different image sizes, section sizes and block rows.

Image Resolution	SECTION_SIZE,(TILE_SIZE, BLOCK_ROWS)			
	256,(32,8)	*256,(32,16)*	*512,(32,8)*	*512,(32,16)*
1024×1024	20.6652	20.8894	18.6917	18.544
2048×2048	21.7337	21.5566	19.9407	19.8363
4096×4096	21.1263	21.0277	19.3964	19.3299
8192×8192	20.9987	20.9943	19.2776	19.2203

Table 4: Speedup achieved by the CUDA C++ optimized implementation tested with different image sizes, section sizes and block rows.

Table 2 shows the speedup obtained relative to the sequential implementation. The improvements are considerable, with values exceeding 14x for large images.

Moving to the optimized implementation, Table 3 presents the execution times achieved. Compared to the naive version, a significant reduction in time is observed. For example, for an 8192×8192 resolution, the minimum execution time decreases from approximately 12.10 ms in the naive version to around 8.38 ms in the optimized version.

Table 4 displays the speedup achieved with the optimized version compared to the sequential implementation. The values exceed 19x in all tests, with peaks above 20x for higher resolution images.

The graph in Figure 6 confirms these observa-

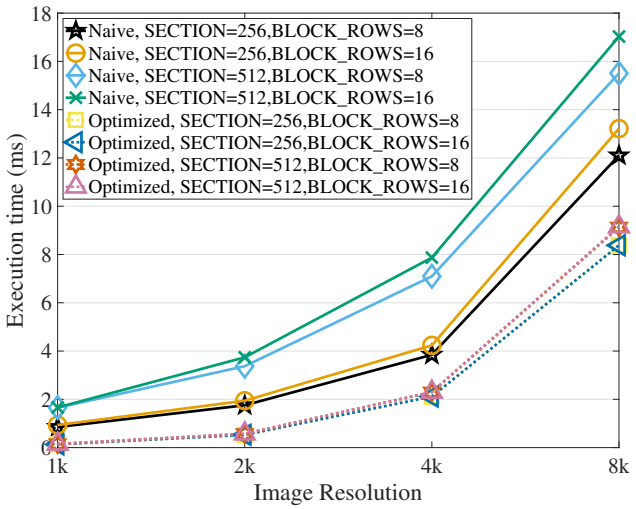

Figure 6: Graph showing the execution times recorded for different image sizes, section sizes and block rows in C++.

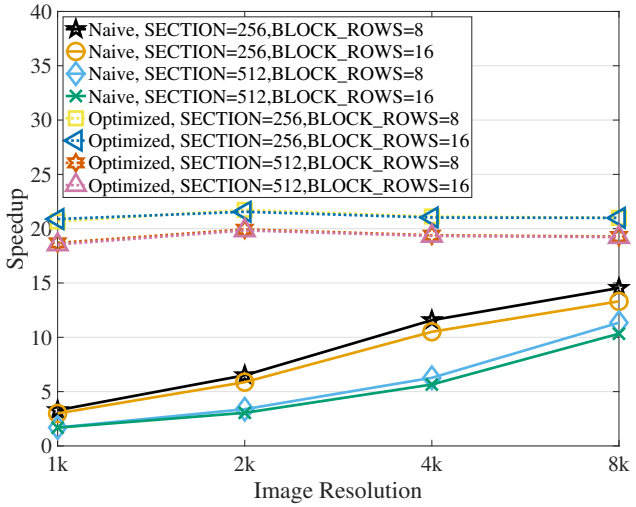

Figure 7: Graph showing the speedups recorded for different image sizes, section sizes and block rows in C++.

tions, showing a comparison between the execution times obtained from both the naive and optimized versions. Figure 7, on the other hand, provides a comprehensive overview of all the speedups.

3.2. PyCUDA Benchmarks

The execution times for the PyCUDA implementation are reported in Table 5. Although this implementation is less optimized compared to the native CUDA C++ version, it still shows a significant improvement over the sequential implementation, with processing times remaining close to those of the native version. In particular, for an image with a resolution of 8192×8192, the minimum execution time is approximately 8.49 ms, a value comparable to that of the native CUDA C++ version.

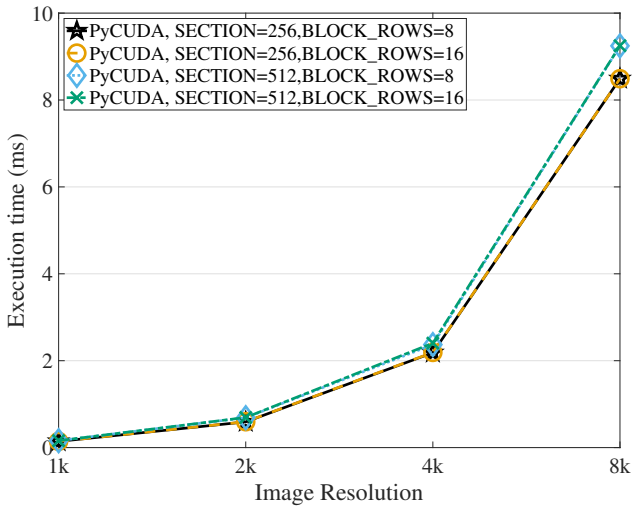

Figure 8: Graph showing the execution times recorded for different image sizes, section sizes and block rows in PyCUDA implementation.

Figure 8 shows the trend of execution times as a function of image size, confirming the efficiency of PyCUDA, while still being slightly less performant than the native version.

3.3. Numba Benchmarks

The results of the Numba-based version are reported in Table 6. In this case as well, reduced execution times are observed compared to the sequential version, although they are slightly higher than those of the PyCUDA and CUDA C++ versions. For an image of size 8192×8192, the minimum execution time is approximately 8.97 ms, slightly higher than that of PyCUDA.

Figure 9 illustrates the trend of execution times for different image sizes, showing behavior similar to that of the other GPU-based implementations.

Image Resolution	SECTION_SIZE,(TILE_SIZE, BLOCK_ROWS)*			
	256,(32,8)	*256,(32,16)*	*512,(32,8)*	*512,(32,16)*
1024×1024	0.142801	0.143320	0.158473	0.159226
2048×2048	0.592311	0.595200	0.688412	0.693234
4096×4096	2.18445	2.19059	2.37036	2.40427
8192×8192	8.48956	8.49030	9.24645	9.25027

*All times are expressed in milliseconds.

Table 5: Execution times for the PyCUDA implementation tested with different image sizes, section sizes and block rows.

Image Resolution	SECTION_SIZE,(TILE_SIZE, BLOCK_ROWS)*			
	256,(32,8)	*256,(32,16)*	*512,(32,8)*	*512,(32,16)*
1024×1024	0.194961	0.196428	0.212179	0.214311
2048×2048	0.654800	0.668436	0.758527	0.76684
4096×4096	2.29498	2.33047	2.49479	2.51469
8192×8192	8.96984	9.05188	9.74185	9.79483

*All times are expressed in milliseconds.

Table 6: Execution times for the Numba implementation tested with different image sizes, section sizes and block rows.

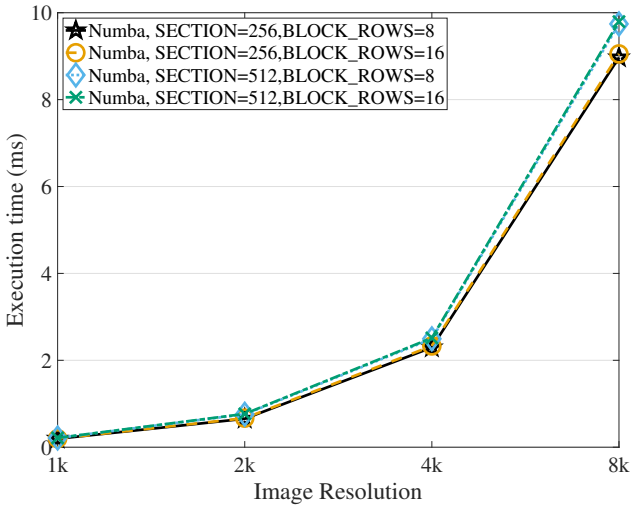

Figure 9: Graph showing the execution times recorded for different image sizes, section sizes and block rows in Numba implementation.

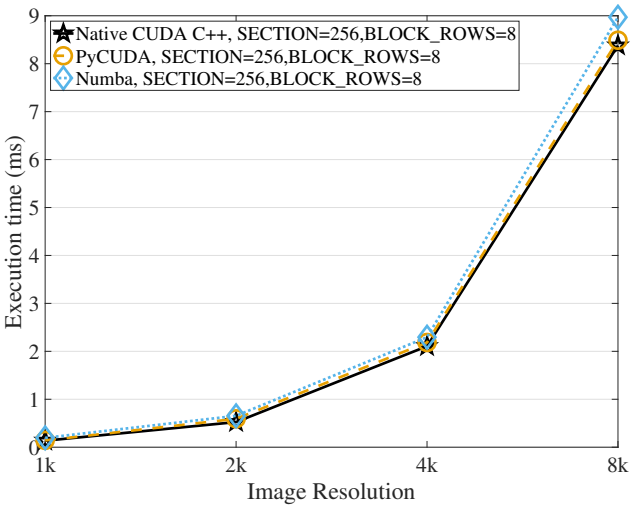

Figure 10: Graph showing the execution times recorded for SECTION_SIZE=256, TILE_DIM=32, BLOCK_ROWS=8 in Native CUDA C++, PyCUDA, and Numba implementations.

3.4. Comparison of Native CUDA C++, PyCUDA and Numba Benchmarks

Table 7 summarizes the execution times of the three GPU-based implementations with SECTION_SIZE=256, TILE_DIM=32, and BLOCK_ROWS=8, compared to the sequential implementation. It is clearly observed that the native CUDA C++ version is the fastest, followed by PyCUDA, and finally Numba. However, the differences between the three GPU versions are relatively small, especially for large images.

The graph in Figure 10 confirms this trend, showing that all GPU implementations significantly reduce execution times compared to the sequential version, with the native CUDA C++ version offering the best overall speedup.

Image Resolution	Sequential*	Native CUDA*	PyCUDA*	Numba*
1024×1024	2.80113	0.135548	0.142801	0.194961
2048×2048	11.4013	0.524591	0.592311	0.654800
4096×4096	44.5048	2.1066	2.18445	2.29498
8192×8192	175.950	8.37909	8.48956	8.96984

*All times are expressed in milliseconds.

Table 7: Execution times recorded for SECTION_SIZE=256, TILE_DIM=32, BLOCK_ROWS=8 in Native CUDA C++, PyCUDA, and Numba implementations.

4. Conclusions

In conclusion, all GPU versions provide a significant improvement over the sequential implementation. The choice between CUDA C++, PyCUDA, and Numba will depend on the specific needs of the user: CUDA C++ ensures the best performance, while PyCUDA and Numba offer simpler integration with Python, with still very high performance.

References

[1] 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. 4

[2] Numba Developers. Numba documentation. `https://numba.readthedocs.io/en/stable`.

[3] PyCUDA Developers. Pycuda documentation. `https://documen.tician.de/pycuda`.

[4] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in cuda. Technical report, NVIDIA Corporation, June 2010. 6

[5] Wikipedia. Summed-area table. `https://en.wikipedia.org/wiki/Summed-area_table`.