Concurrency and Parallelism Project

Guilherme Fernandes
Vladyslav Mikytiv
60045
60735

ABSTRACT

This report provides the explanation and the implementation of parallelisation of the algorithm of equalization of images. Both an OpenMP and CUDA implementations.

KEYWORDS

OpenMP Parallelism C++ CUDA

1 CRITICAL CODE ANALYSIS AND PARALLELIZATION

1.1 The first steps

At first we ran the code without doing any modifications and used the profiler in order to identify the hot spots doing a average value in each image of the time lost in functions from the profiler. The algorithm runs for 100 iterations.

Function	Computation Time Lost (%)
correctColor&rescale	57.98%
grayScale	23.17%
normalize	18.24%

Table 1: Average Computation Time Lost in Functions

As seen in Listing 1 the zones of the code that took a long time to perform the computation and in overall slow down the algorithm are: the **`normalization`** of the image, the **`correction of color`**, the **`convertion to the greyscale`** and some other not so computational heavy functions.

To reduce the impact on processing time caused by these functions within the algorithm, we integrated OpenMP directives into these specific parts of the code to accelerate computation.

It's important to highlight the **thread management**. Since we are working with images we will have to do some simples calculus to determine the amount of work that each thread will be given. For that we must perform two calculations (one for RGB and one for the grey scale). For the RGB we will calculate `WIDTH * HEIGHT * 3` and to obtain the `CHUNK_SIZE_RGB` we just divide `WIDTH * HEIGHT * 3` by `N_THREADS`. For the grey scale it's similar but we don't multiply by 3 getting the `CHUNK_SIZE`. Now we will have the work that will be balanced between threads.

1.2 Function parallelization

The **normalization** function will be improved with the following code: **`pragma omp parallel for`** with the option **`schedule(static, chunk_size_channels)`** and **`num_threads(n_threads)`**. This code optimization can be seen in Listing 1.

```
void normalize(//omitting for space) {
    #pragma omp parallel for schedule(static,
     chunk_size_channels) num_threads(n_threads)
    for (int i = 0; i < size_channels; i++)
        uchar_image[i] = (unsigned char) (255 *
     input_image_data[i]);
}
```

Listing 1: Normalization Function

The **grey scale conversion** will be improved in two ways. We will mix the `fill_histogram` function with this one in order to do everything in the same function. Besides that we also apply the following OpenMP directives: **`pragma omp parallel for`** with the option **`reduction(+:histogram)`** and **`num_threads(n_threads)`**. This code optimization can be seen in Listing 2.

```
  void convertoToGrayScale(//omitting for space) {
        // filling the histogram with zeroes
        #pragma omp parallel for reduction(+:histogram)
     num_threads(n_threads)
        for (int i = 0; i < size; i++){
                auto r = uchar_image[3 * i];
                auto g = uchar_image[3 * i + 1];
                auto b = uchar_image[3 * i + 2];
                gray_image[i] =
                static_cast<unsigned char>
                (0.21 * r + 0.71 * g + 0.07 * b);
                histogram[gray_image[i]]++;
            }
    }
```

Listing 2: Grey Conversion Function

The function that **calculated the CDF** doesn't require any type of parallelization. It's a simple for loop that executes 256 iterations every time. Another function that we managed to simplify was the `cdf_min_loop`. Since we minimum will always be on the first position we just return it and that's how we compute the minimum of the CDF.

The **`correct_color_loop`** can be mixed with the **`rescale`** following the same strategy as we did with `grey scale` and `fill historgram` functions. The code optimization is the same as Listing 1 and can be seen in Listing 3.

```
  void correct_color_loop_and_rescale(//omitting for
     space) {
        #pragma omp parallel for schedule(static,
     chunk_size_channels) num_threads(n_threads)
        for (int i = 0; i < size_channels; i++)
        {
            uchar_image[i] = correct_color(cdf[
     uchar_image[i]], cdf_min);
            output_image_data[i] = static_cast<float>(
     uchar_image[i]) / 255.0f;
        }
    }
```

Listing 3: Color Correction Function

1.3 Decions

It's important to justify our reasoning. We used **`static`** in every single call of the OpenMP directives because the extra overhead of orchistrating the threads with a certain amount of work did not compensate. As we can see in the Table 3 **`static`** always shows better results and it's coherent to the theoretical analysis. For this simulation we used 16 threads.

Dynamic Time (ms)	Static Time (ms)	Image Name
117	87	borabora.ppm
98	81	input01.ppm
5620	4334	sample.ppm

Table 2: Dynamic vs Static Mean Execution Times

2 METRIC ANALYSIS - OPENMP

Now, let's assess the impact of these changes on the runtime of our program. For that we will use **speed up** and **efficienty** and we will execute every executing multiple times in order to get the mean value of the execution times.

The code was in a cluster executed with the following specifications: **2x Intel Xeon E5-2609 v4**, with **16 cores** and **NVIDIA Quadro M2000** GPU.

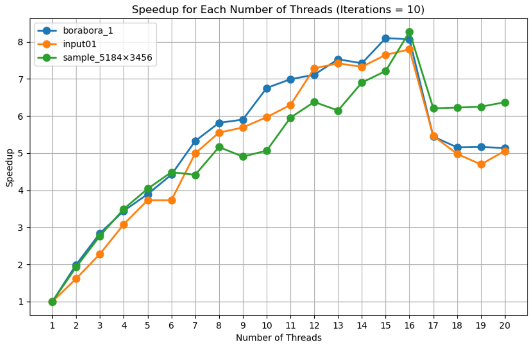

Figure 1: Speedup across different number of threads usage of different images

Figure 1 describes the speed up for the different images in our test set. We can clearly see a rise on the speed up until a certain point. After that point the value of the speed up begins to decrease. The optimal `NR_THREADS` to perform the computation is 16.

In order to calculate the efficiency we will use the best result that we had: the one with 16 threads. For that we define S_{16} which is the speed up with 16 threads and it will be equal to

$$S_{16} = 8$$

And the efficiency of the parallelel implementation in Section 1 get's us an efficiency value of:

$$E_{16} = \frac{S_{16}}{16} = 50\%$$

3 CUDA

We've attained speed enhancements leveraging the OpenMP library for parallelizing our code. However, there exists an avenue to further escalate our computational efficiency. By harnessing the GPU architecture through CUDA, we can unlock additional acceleration potential.

To leverage the GPU effectively, we need to define kernels for specific regions of our code that we wish to offload to the GPU for computation. This necessitates defining grids use in CUDA kernels.

Utilizing the GPU for optimization involves creating kernels to transfer data to the GPU, perform computations, and retrieve the results back to the CPU. Given the high cost associated with these operations, minimizing the number of kernels is imperative. After several iterations, we've determined that the minimum number of kernels required is three. Due to dependencies, namely the need for histogram calculation in operations like correcting colors and resizing, we cannot combine the first kernel with the third one.

The initial kernel handles both the normalization and grayscale operations. Since these operations are independent of each other, we can consolidate them into a single kernel instead of creating two separate ones. The Listing 4 shows this kernel implementation.

```
__global__ void normalize_kernel(//omitting for space) {
        int ii = blockIdx.y * blockDim.y + threadIdx.y;
        int jj = blockIdx.x * blockDim.x + threadIdx.x;
        int idx = (ii * width + jj)*3;

        if (ii < height && jj < width) {
            for(int i = 0; i < 3; i++) {
                // Normalize logic
            }
            auto r = uchar_image[idx];
            auto g = uchar_image[idx + 1];
            auto b = uchar_image[idx + 2];
            idx =ii * width + jj;
            gray_image[idx] = // Calculation
        }
    }
```

Listing 4: Grey Conversion and Normalization

Before proceeding to the second kernel, it's essential to calculate the histogram. While we could create a custom kernel for this task, a more efficient approach involves leveraging the **`CUB`** library. By utilizing its operations, we can significantly enhance the performance of our code.

By using **`cub::DeviceHistogram::HistogramEven`** we are able to calculate the histogram in the GPU without making the kernel ourselves.

After having the histogram we can calculate the probabilities in order to use them in another call to the **`CUB`** library. For that we create a second kernel, which will seem more clear why ahead. This one will differ from the first one in the **`numBlocks`** and **`blockSize`**. The **`numBlocks`** will be 256 and the **`blockSize`** will be `(HISTOGRAM_LEN+ blockSize-1) / blockSize`.

Once we have computed this probability array, we can further utilize it in another function provided by the **`CUB`** library.

The **`cub::DeviceScan::InclusiveSum`** function facilitates the calculation of the final cumulative distribution function (CDF) values. This not only streamlines the programming process but also optimizes computation time, offering significant efficiency gains.

We used the **InclusiveSum** because it was not possible to use **cub::DeviceScan::InclusiveScan** due to restrictions in the library (there was no way to pass the probabilities to this call). That's why we must calculate the probabilities beforehand and use the **InclusiveSum**.

Ultimately, we employ our final kernel, which could theoretically be executed within the first kernel. However, due to dependencies necessitating separate execution, we're compelled to create this third kernel to handle the last part of the algorithm. The code can be seen in Listing 5.

```
 __global__ void correct_kernel(//omitting for space) {
        int ii = blockIdx.y * blockDim.y + threadIdx.y;
        int jj = blockIdx.x * blockDim.x + threadIdx.x;
        int idx = (ii * width + jj)*3;

        if (ii < height && jj < width) {
            for (int i = 0; i < 3; i++) {
                auto cdf_val = d_cdf[uchar_image[idx+i]];
                float cdf_min = d_cdf[0];

                uchar_image[idx+i] = // Logic

                output_image_data[idx+i] = // Logic
            }
        }
    }
```

Listing 5: Correct and Rescale

These kernels constitute the entirety of the GPU-accelerated operations. Due to space constraints, significant portions of the code have been omitted. For the complete implementation, please refer to the accessible code at https://github.com/Gui28F/CP-Project.

4 METRIC ANALYSIS - CUDA

Utilizing the capabilities of the GPU necessitates defining the grid dimensions. To determine the optimal TILE_WIDTH value, we conducted an experiment, the results of which are depicted in Figure 2. It was discerned that the most favorable value was 16. Notably, we are constrained from exceeding 32 due to the limitation imposed by the maximum thread utilization, capped at $32 * 32 = 1024$.

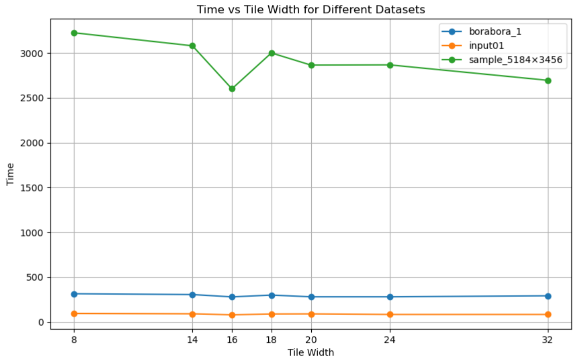

Figure 2: Tiles Width and Execution Time

To effectively evaluate the performance of the GPU-accelerated version, it's important to compare it against the most optimized configuration achieved with OpenMP (specifically, the one utilizing 16 threads). Figure **??** illustrates the differences in speedup compared to the optimal OpenMP configuration.

5 FINAL RESULTS

After implementing optimizations using **OpenMP** and **CUDA**, we now have a clearer understanding of the overall performance. The table presents execution times for the purely sequential implementation, the **OpenMP** implementation, and the **CUDA** implementation. These results, obtained from 100 iterations of the algorithm, represent the mean calculated values derived from running the algorithm five times.The names were shortened in the table [1].

SQT (ms)	OMPT (ms)	CUDAT (ms)	Image
742	84	92	borabora.ppm
619	80	65	input01.ppm
29600	4320	2600	sample.ppm

Table 3: Final Execution Times

6 CONCLUSIONS

Whether through the integration of OpenMP directives or harnessing the power of GPU acceleration, we consistently observe improved results compared to the raw sequential implementation. Notably, in the case of the borabora.ppm image, an anomaly surfaces where the OpenMP implementation slightly outperforms the CUDA counterpart. We attribute this discrepancy to potential pixel saturation issues. Nonetheless, it's important to note that, in general, the CUDA implementation remains the preferred choice, with ongoing comparisons against OpenMP to ensure optimal performance.

7 CODE

All the code mentioned above can be consulted in https://github.com/Gui28F/CP-Project.

8 INDIVIDUAL CONTRIBUTION AND COMMENTS

The project was a joint effort, with both team members sharing the workload equally. We utilized pair programming sessions on Discord calls and in the lab to accomplish our tasks effectively. We are grateful to Professor Hervé for his invaluable support and constant availability to assist us throughout the project.

[1] **Description:** SQT – Sequential Time, OMPT – OpenMP Time, CUDAT – CUDA Time