15-418 Spring 2020 Final Project Report

Parallelizing Viola-Jones Facial Detection Algorithm

Iris Wang (imwang), Minji Kim (minjik4)

Summary

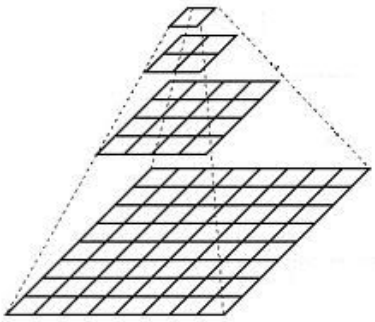
We implemented the Viola-Jones algorithm in CUDA on GPU and in OpenMp on the CPU. We optimized the program and compared the resulting performance of the two implementations. We analyzed the algorithm and identified potential places of parallelism. We managed to achieve around a 5x speedup on OpenMp and a 20x speedup on CUDA.

Background

The Viola-Jones Object Detection Framework is a fast, sequential framework for general object detection. This framework is particularly successful in rapid facial detection when used with Haar features. The algorithm can be decomposed into two major steps: training and detection. For the purposes of this project, we use pre-trained parameters to create classifiers used in the detection step.
The detection step consists of several steps shown in the pseudo code below. Each step will be further discussed in detail.

```
for number of scales in image pyramid:
    downsample image by one scale
    compute integral image for new scale
    for each shift step of the sliding detection window:
        for each stage in the cascade classifier:
            for each filter in the stage:
                filter the detection window
            end
            accumulate filter results for this stage
            if accumulation fails to pass this stage's threshold:
                break the for loop and reject this window as a face
            end
        end
        if this detection window passes all per-stage thresholds:
           accept this window as a face
        else:
           reject this window as a face
        end
    end
end
```

Image Pyramid



An image pyramid is used to use the same detection window to detect variable size faces within an image. This is more efficient than trying to resize the detection window, thus many computer vision algorithms use this technique. For each scale, the image is downsampled using neighboring pixels. This is implemented in the function **nearestNeighbor**.

Figure 1.1: Image pyramid

Integral Image

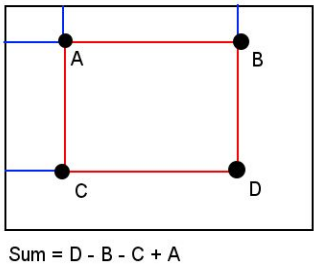


Figure 1.2: A description of computing a sum using an integral image

The integral image is an efficient way to find a sum of values within an arbitrary rectangular region defined by its four corners. One can create an integral image by computing the sum of all the pixels above and to the left of each pixel. After computing the integral image, the sum of values within a rectangular area with (A, B, C, D) as corners is simply equivalent to D - B - C + A, as shown above in Figure 2. The information about the integral image is stored in **MyIntImage** structure consisting of width, height, and an 1-D integer array of values at each point in the image.

```
typedef struct
{
  int  width;
  int  height;
  int* data;
  int  flag;
}
MyIntImage;
```

Figure 1.3: **MyIntImage** struct definition

Sliding Detection Window

The detection window is used to “slide” across the entire image to detect faces at each location. The detection window in our implementation is shifted by one pixel at a time. The image region selected by each detection window will be evaluated using the cascade classifier.

Cascade Classifier

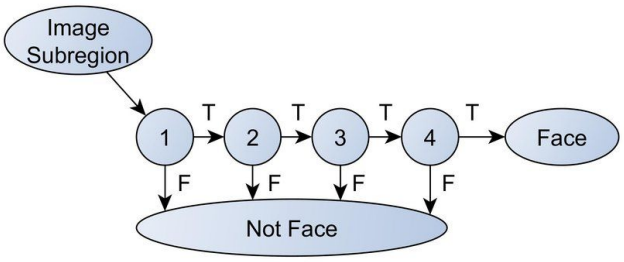


Figure 1.3: Illustration of a typical cascade classifier routine

The cascade classifier is a serial combination of multiple sets of filters. Each set of filters is called a “stage”. Each subregion selected by a detection window will go through the stage sequentially, and it will reject the subregion if any one of the stages fail. If the subregion passes all of the stages, it will be marked as a face. The filters in each stage are Haar filters, which detect Haar-like features in an image. For this implementation, the information about the filters from the pre-trained classifier is stored in **class.txt**. The function **readTextClassifier** will parse the filter parameters and store them in 8 different 1-D integer arrays as shown in Figure 1.4.

```
static int *stages_array;
static int *rectangles_array;
static int *weights_array;
static int *alpha1_array;
static int *alpha2_array;
static int *tree_thresh_array;
static int *stages_thresh_array;
static int **scaled_rectangles_array;
```

Figure 1.4: Data Structures storing cascade filter information

Sources of Parallelism

The three main parts of this algorithm that are computationally heavy are:

(1) computing the integral image

(2) evaluation of each sliding detection window across the image

(3) going through all the stages in a cascade classifier.

The computation of the integral image is by performing a sequential one-pass through the matrix. Although this process isn't extremely time-consuming on modern processors, the computation time will get worse linearly as the image size grows. This sequential process can be parallelized in multiple ways by having threads perform sums in each row, columns, or both. We will discuss the details of different approaches to parallelizing this computation in the later section.

Evaluation of each sliding window across the image is another major source of parallelism. Since each window is independent of each other, it is a reasonable approach to parallelize over the windows and have multiple processes evaluate multiple windows in parallel. This will greatly reduce the overhead created by the sequential processing of each window, especially as the number of detection windows to cover the entire image significantly increases for smaller window sizes.

The process of going through the cascade classifier for each subregion also presents a source of parallelism. Since each stage's filters are independent of each other, it is possible to parallelize over the stages, where each process takes over a stage. However, this will remove the ability to terminate the evaluation of the classifier when any one of the stages fail. This means that the parallelized program may be subject to spending time on unnecessary computations, since each process will be unaware of any of the earlier stages failing.

Approach

Technologies

In this project, we will take a working C++ implementation of the Viola-Jones algorithm with pre-trained classifiers and create parallel implementations using two different programming environments: CUDA and OpenMP. A lot of the sequential code has been maximally optimized to provide a reasonable performance, thus we targeted the sections that will benefit the most from parallel computation. We focused on strategies to parallelize the three major sources of parallelism discussed in the previous section using the two environments.

Input Images

The primary image we used for testing correctness of our program was **big-many-1.ppm**, which is a relatively larger image with more faces. For benchmarking purposes after finalizing our parallel implementations, we used several images that vary in two key attributes: image size and the number of faces. We created a image set consisting of different combinations of the two attributes and used them to compare and contrast performance between OpenMP and CUDA implementations. Images were found through Google Photos, where some of them were converted to PGM format to be used in our implementation. Below is a summary table of the information about the input image files used.

Input File (.pgm)	Width (px)	Height (px)	# of Faces
small-few-1	225	224	1
small-few-2	750	500	2
small-many-1	450	300	9
small-many-2	256	256	7
big-few-1	1071	1260	4
big-few-2	1300	827	4
big-many-1	1280	626	29
big-many-2	1200	960	10

Table 2.1: Input images used for testing.

Below are some examples of the output images after running the program on the given input image. White rectangles indicate the detected faces.

Figure 2.2: Output of big-many-2.pgm

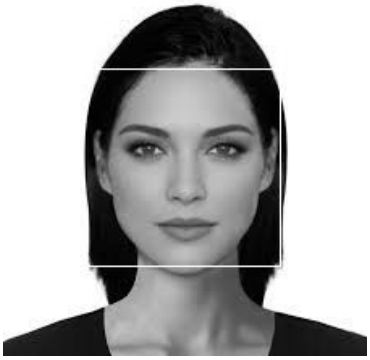


Figure 2.3: Output of small-few-1.pgm

Machines

For running the OpenMP implementation, we used the 8 CPU cores in GHC cluster machines. The same machine was used to run the original sequential implementation under a single core. Since the GHC cluster machines have 8 cores and none of them support hyperthreading, the maximum number of OpenMP threads we could utilize is 8. For running the CUDA implementation, we used the NVIDIA GPUs connected to the GHC cluster machines. The specific model information is stated below.

CPU	Intel(R) Core(TM0 i7-9700 CPU @ 3.00GHz
GPU	GeForce RTX 2080

Table 2.4: Machines used for running parallel implementations

OpenMP Implementation Details

1. *Parallelizing Integral Images*

We started out by parallelizing the computation of integral images. The computation-heavy part of this process is the sequential summation of pixel values for each pixel in the scaled image. The biggest limitation was that the algorithm consisted of performing a left-to-right, top-to-bottom pass through the image to accumulate the summed value for each pixel. Therefore, computing the summed values is data-dependent on the previously pixels. In order to remove the data dependency of pixels, we decided to modify the sequential algorithm. Instead of doing a single pass through the image, we can first perform a row-wise summation and then perform a column-wise summation. This will have the same effect of summing up all the pixels above and to the left at each point. By modifying the algorithm to perform two separate passes, we are now able to perform each pass in parallel. To implement this in OpenMP, we split up the function into two main for loops and attached **#pragma omp parallel for** for both loops.

The row-wise summation was straightforward to implement in OpenMP, since each thread is accessing pixels in the same row, leading to minimal memory access overhead. However, for the column-wise summation, each thread has to access pixels across all of the rows, which results in significant cache misses and false sharing among the threads. Taking this memory access overhead caused by column-wise summation into consideration, we devised two potential implementations.

The first implementation was to transpose the matrix before and after the column-wise summation. This will allow the column-wise summation to just be another row-wise summation, removing the extra memory access overhead. However, this also meant that there will be more work being done to transpose the matrix twice. To minimize the computation time for transposing the matrix, we decided to parallelize the transpose function using OpenMP as well. After a number of iterations, we were able to come up with the optimal transpose function. We split up the matrix into multiple submatrices, where each OpenMP thread transposes each submatrix. This blocked transposing function allows us to maximize the cache use for each core, since transposing a matrix requires a process to not only access elements in the same row but also in the same column. Therefore, by operating on a smaller matrix, each thread is able to keep all the necessary elements in the cache without evicting elements that are needed later.

The second implementation was to perform a column-wise summation without transposing at all. This will still have the issue of delayed memory access when performing a column-wise summation, but it won't be necessary to perform extra work of transposing.

After implementing both versions and comparing the performance to the sequential performance, we found that the first implementation achieved trivial speedup. However, the second implementation achieved at most 2x speedup compared to the sequential version. This is because the computational overhead from transposing a matrix two times for bigger image sizes was significant. This overhead overrode the benefit of performing a row-wise array access instead of column-wise access. Thus, we decided to continue with the second implementation.

Another problem we noticed was that the parallelized algorithm only achieves significant speedup when the image size is large. The sequential algorithm was faster for images that are smaller (due to higher scale factors). In order to solve this, we set a certain threshold for the image size to decide whether to run the sequential or the parallel version to compute the integral image.

2. Parallelizing Over Sliding Detection Windows

The most computational-heavy part of the algorithm is from the evaluation of the cascade classifiers. The function **ScaleImage_Invoker** loops through each window that is shifted by one pixel, and evaluates the subregion using the cascade classifier. We noticed that the same cascade classifier is applied to every window, and it is possible to parallelize the process since the evaluation of each window is independent. We started by inserting **#pragma omp parallel for** on the two for loops that shifted the window

horizontally and vertically. However, the problem was raised when a face is detected in the middle of each process. All the information regarding the detected faces are stored in a global C++ vector. Adding an element to this vector upon a face detection requires the algorithm to perform a **push_back** operation, which is not thread-safe. To resolve this issue, we decided to protect the operation with **#pragma omp critical**. We speculated that overhead caused by this critical section was minimal since the number of times a window will detect a face is significantly lower than the number times a window will be rejected. As a result, we achieved roughly 7x speedup on the evaluation of classifiers across all windows by parallelizing the process over each sliding detection window.

```
// if a face is detected, record the coordinates of the filter window
if( result > 0 )
{
  MyRect r = {myRound(x*factor), myRound(y*factor), winSize.width, winSize.height};

  #pragma omp critical
  vec->push_back(r);
}
```

Figure 2.5: Defined a critical section when performing vector operations.

Another optimization strategy for parallelizing over windows was to utilize different scheduling tactics for pragma omp loops. We found that the maximal speedup is achieved when we use dynamic scheduling for the outer for-loop (horizontal shift) and static scheduling with a chunk size of 64 for the inner for-loop (vertical shift). This is an expected behavior, since each iteration of the outer for-loop have varying execution times depending on the window. If the iteration contains a face, the execution time will be much longer than the iteration without a face. Thus, to minimize the load imbalance between the threads, we use dynamic scheduling on the outer loop. For the inner loop, we statically scheduled the threads with a larger chunk size to exploit spatial locality. Such blocking of the window allows each thread to maximize the cache use and reduce the memory access overhead.

```
#pragma omp parallel for schedule(dynamic)
for( int x = 0; x <= x2; x += 1 ) // horizontal shift of window
{

  #pragma omp parallel for schedule(static, 64)
  for( int y = 0; y <= y2; y += 1 ) // vertical shift of window
  {
```

Figure 2.6: Utilization of different scheduling in pragma omp for loops

Since the computation time spent on this part of the algorithm was dominating the overall execution time of the program, parallelizing this function led to greater overall speedup then parallelizing the computation of integral images. In contrast, the 2x speedup in the integral images function only led to trivial speedup in the overall execution time. Nonetheless, since the integral images were still significantly faster than the sequential version for larger scaled images, we decided to keep both parallelization methods for our final OpenMP implementation.

3. Parallelizing Over Stages in a Cascade Classifier

A different approach we took to speed up the process of the cascade classifier evaluation was to parallelize each stage. We removed the window-wise parallelization implemented above, and re-implemented the function to only parallelize over stages. As described in the earlier section, the cascade classifier consists of multiple stages. The sequential algorithm goes through these stages one-by-one, rejecting the window immediately when any one of the stages fail. We tried to parallelize this process by inserting a **#pragma omp parallel for** construct for the section that was looping through each stage. However, this meant that we can no longer quit out of a window evaluation when earlier stages fail. As a result, we found that the performance is actually worse when we parallelize over stages, due to all the unnecessary computation of extra stages that would have been omitted from the evaluation process. Therefore, we concluded that the optimal strategy is to optimize over the windows and leave the stages to be evaluated sequentially.

CUDA Implementation Details

1. *Parallelizing Integral Images*

We started by attempting to parallelize computing the integral image for each scale. We parallelized this process again by splitting up into two separate passes through the array. First, we copied over the data array containing pixel values to the device memory. Then, we summed up all the rows in parallel, and then summed up all the columns in parallel to get the final integral image sum array on the GPU. Finally, we copied the integral image array back over to the host memory.

Unfortunately, this did not lead to the speedup we were expecting since the computation of the integral image is not a significant part of the execution time in the sequential implementation. We measured the execution times of each critical step of the algorithm in the sequential version and realized that the time spent in integral image was insignificant compared to the image processing step. We tried to incorporate this improvement with our window parallelization, however it slowed down our performance. This is because the overhead that comes with launching the kernel and copying the large array needed for calculating the integral image is not mitigated by the m speedup that CUDA gives. The sequential version already does the calculation relatively quickly enough, and it is optimal to keep it in our final version of our CUDA implementation.

2. *Parallelizing Over Stages in a Cascade Filter*

Another optimization method that we attempted was parallelizing over the stages in the cascade classifier as described in the OpenMP section. For each sliding detection window, we mapped a thread to a stage, so that each thread would process each stage and its respective filters in parallel. Each thread then iterates through the number of filters for the stage it was assigned (based on the thread id). Then, we accumulate the results and reject or accept the window as a face based on the result array written by all the threads. However, this also did not lead to the speedup we wanted. First, we are not utilizing all the threads possible, as we are limited to as many threads as we have cascade stages (25 for our pre-trained classifier). Second, a window can be rejected at any stage, but running it in parallel means we have to wait for all the stages to finish computing, leading to unnecessary computation. Each stage also does not have an equal amount of filters, so the running time for each thread would vary greatly, causing synchronization overhead.

3. Data Structure Optimizations

Some optimizations we made along the way were changing the sequential code from utilizing int** arrays to int* when passed to the GPU. We realized that it's more expensive memory accesses than just using a 1D array since it requires two memory accesses to global memory, thus slowing the program down. We also used a local variable to add up each stage sum for each thread in order to reduce false dependency.

4. Parallelizing Over Windows

Our final CUDA implementation parallelizes the algorithm over windows, similarly to the final OPen MP implementation. Since the same cascade filter is used each time and each filter step is independent, we performed those calculations in parallel for each shift step of the sliding detection window of the image. We performed the cascade classifier evaluations of a facial feature on the GPU, where each thread was responsible for one sliding window of the image. We spawned image size divided by number of threads (256) blocks for each kernel launch, so that there each thread would be mapped to a window in the image. Then, each thread would process all the stages in the classifier, and run each filter in each stage in parallel. Each thread would add up the sum of processing each stage and the filters in the stage for their respective window. It would then check if their stage sum matched the facial feature identifier threshold and if it did, write a 1 in the GPU results array, indexed by their respective thread id and image size in order to reduce memory access conflicts. After all the threads finished processing their respective window, we would sync all the threads and copy the results back to the CPU. With that results array, we would be able to cycle through and identify which points in the image corresponded to a facial feature.

Originally, in the serial code, each point in the image would be passed through the cascade stages (and multiple filters) and at each stage, the algorithm would check if it has passed the threshold values that would indicate it is a facial feature and reject or accept it based on those results. However, we had to modify the algorithm in order to remove any data dependencies in this step so that the rejection of a point would happen after all the threads finished processing their respective window.

We launched the current image size divided by the number of threads (256) blocks. We mapped each thread to a sliding window. We chose to have 256 threads for our program as we wanted to keep the size of a thread block a multiple of 32 due to the mapping between warps and thread blocks. We

indexed accesses of the threads to the data arrays by the threadIdx in order to coalesce the memory transactions of the threads. In addition, we utilized loop tiling - the first access of the shared arrays by the first thread loaded them into the cache, where we accessed continuous indices of the arrays, thus taking advantage of spatial and temporal locality. We also used loop unrolling to speed up the program by doing more accesses per iteration.

Results and Analysis

Goals

We reached the goal that we were planning to achieve when we started this project. We achieved upto 20x speedup for CUDA (disregarding kernel launch overhead) and 5x speedup for OpenMP. We measured the performance through wall-clock time, using the built-in Chrono library in C++. We placed timers around critical functions of the program in order to measure time taken for each section and around the entire program to analyze the difference in execution times.

Experimental Setup

Our experimental setup included inputs of pictures of various sizes and number of faces to identify. We varied our input image size from 225 * 225 to 1200 * 960 and with a number of faces ranging from 1 to 29, in order to gather a more coherent analysis of the speedups of our parallel implementation on different sized inputs. We wrote a script that runs the program 10 times on the given image and takes the average of the execution times. We used this for the benchmark data for all implementations. For CUDA, we also measured the execution time with and without the startup overhead, as the overhead doesn't represent the true speedup gained from the parallelized functions in CUDA. In order to calculate the kernel launch overhead, we ran the program with nvprof, which showed us the execution time of CudaMalloc and CudaMemcpy. We removed the time spent on those API calls from the total execution time to calculate the actual execution time of the kernel alone.

Overall Execution Time Analysis

Input File (.pgm)	Sequential (µs)	OpenMP (µs)	CUDA w/ Launch Overhead (µs)	CUDA w/o Launch Overhead (µs)
small-few-1	40,665	14,067	14,067	8,788
small-few-2	375,050	73,951	73,951	17,132
small-many-1	137,641	32,136	32,136	7,624
small-many-2	67,560	18,855	18,855	3,239
big-few-1	1,286,351	243,744	243,744	74,305
big-few-2	799,290	177,922	177,922	39,167
big-many-1	891,598	163,767	163,767	39,027
big-many-2	927,783	185,850	185,850	45,302

Table 3.1: Total Execution Time Comparison for All Implementations (in microseconds)

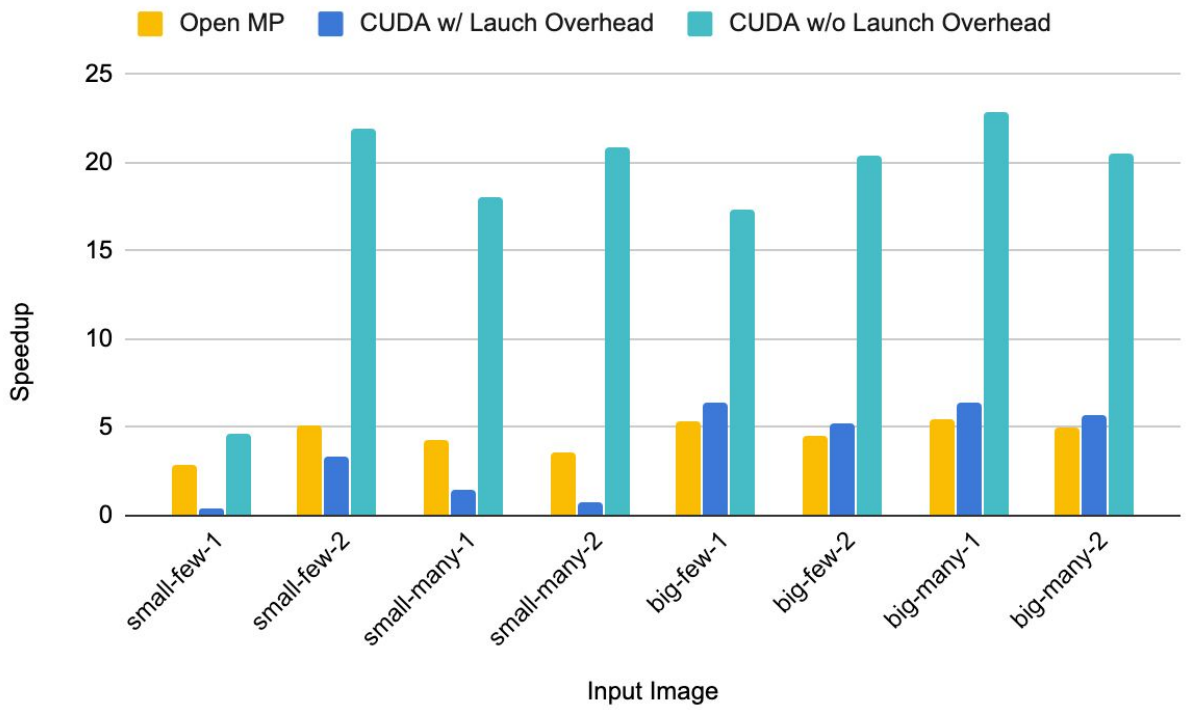


Figure 3.2: Speedup Graph of Parallel Implementations using OpenMP and CUDA

Our baseline is an optimized single-threaded implementation on the CPU. As shown in the chart above, CUDA without launch overhead performed significantly better than OpenMP. This is because without factoring in the cost of allocating space, copying over data to the GPU, and the overall start up time, CUDA can utilize significantly more threads than OpenMP. The degree of parallelization is much higher since in OpenMP, the max number of threads is 8, while in CUDA, we are spawning 256 threads for multiple blocks depending on the image size. The launch overhead is a considerable amount due to the delay from initializing the GPU, copying any executable code, and memory transfers. Therefore, when CUDA with the launch overhead is compared to OpenMP it is often worse or similar in performance.

CUDA speedup with launch overhead for bigger images is similar to OpenMP and for smaller images it is worse. The CUDA speedup for bigger images is significant enough to compensate for the launch overhead due to a higher potential for parallelization, as we spawn the number of threads according to the number of detection windows. There are more windows in a larger image that can be computed in parallel, thus speeding up the program as more threads are being utilized. On the other hand, it is worse for smaller images since smaller images have less windows to be evaluated, thus less possibility for parallelization and the launch time will dominate the execution time. CUDA speedup without considering launch performed the best for all different input workloads, due to the possibility of utilizing hundreds of threads to run the classifier calculations in parallel. The number of faces did not have a noticeable impact on the performance of the parallel implementations. This is due to the fact that rejecting a face early in the stage process does not have an impact on the overall execution time of the program. All the threads must run to completion before moving on, so we wait for the longest stage to finish before accumulating the results and identifying which are faces. This synchronization overhead is one major bottleneck of the parallel implementations of this algorithm when parallelizing over windows.

OpenMP Advanced Analysis

For the OpenMP implementation, we wanted to compare the performance for varying numbers of threads utilized. Below is a graph of the total execution time when the program is run with the input image of **big-many-1.pgm.**

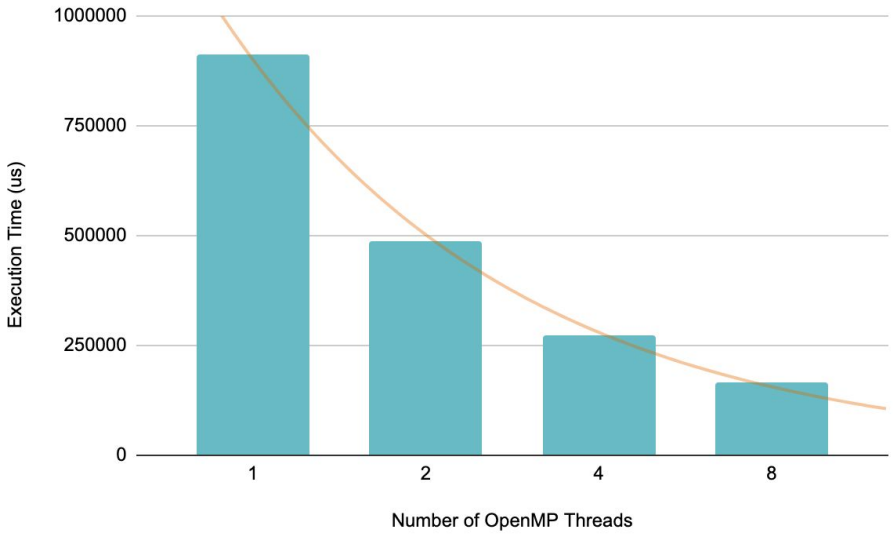


Figure 3.3: Execution time of OpenMP implementation with varying number of threads

As shown in the graph above, the execution time decreases exponentially (orange trendline) as the number of threads increases by a factor of 2. This is a reasonable behavior, given that we optimized the OpenMP implementation to minimize the communication cost through blocked matrix operations and static assignment of threads with bigger chunk sizes. This result shows us that the communication overhead from having more threads is low enough to not limit the performance of our program when it is run with multiple threads.

Another interesting comparison is made between the speedup achieved by parallelizing the integral image and by parallelizing over windows. Since our final OpenMP parallelization incorporates both strategies, we analyze the speedup achieved by each strategy. Below is data of the individual function's execution time when the program is run with the input image of **big-many-1.pgm.**

Source of Parallelization	Sequential Execution Time (μs)	Parallel Execution Time (μs)	Speedup
Integral Images	5,296	2,241	2.36 x
Windows	251,365	33,340	7.54 x

Table 3.4: OpenMP performance data on different sources of parallelization

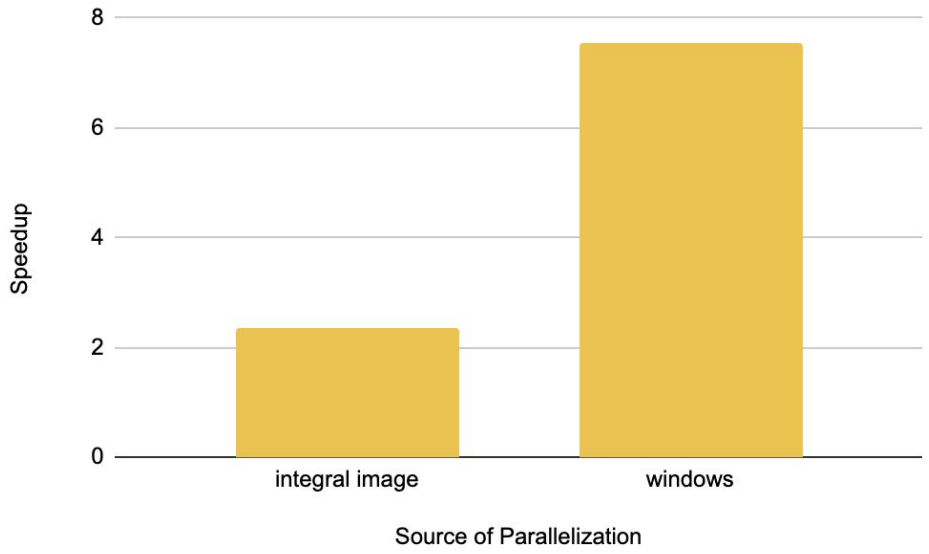


Figure 3.5: Speedup for 2 parallelization strategies used in OpenMP implementation

We see that the parallelization over windows achieved much more speedup compared to the sequential implementation. This tells us that most of the speedup significantly comes from the windows parallelization. This is due to the performance bottlenecks discussed earlier in the integral image parallelization section; the data dependency between the pixels in the integral image limits the performance. On the other hand, each window is completely independent from each other, allowing us to see much more significant speedup when parallelizing over it.

CUDA Advanced Analysis

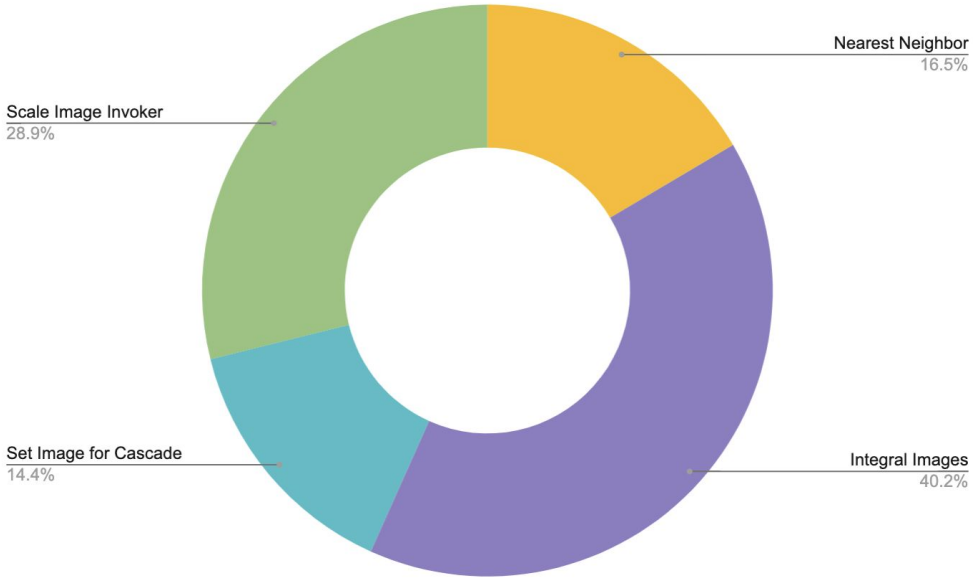


Figure 3.6: Execution time breakdown for CUDA implementation (no overhead)

As we can see from the figure above, time spent in the four functions are relatively similar, with Integral Images and Scale Image Invoker (the cascade classifier) having slightly longer execution times than the other functions. There is room to improve in both Nearest Neighbor and Integral Images, although it would have to be parallelized with fewer kernel launches than our recent approach to Integral Images. We may be able to change the accesses and improve on the spatial locality of calculating the integral image. The main bottlenecks of the performance currently are on the overhead regarding CUDA start up time, which are just a limitation of using the GPU.

Limitations

Our speedup was limited by both lack of sources of parallelism in the original algorithm and also by data transfer overhead for CUDA implementation. This algorithm overall has a sequential flow of work, so we only had a few select functions that could benefit from parallelization rather than the entire process. We must first build the image pyramid, compute the integral image for each scale of the pyramid, and then process each window through the cascade classifier. The process itself cannot be parallelized so we had to focus on parallelizing individual steps of the process. In addition, our speedup for CUDA was limited by the amount of data that we had to transfer to the GPU. The processes that we were parallelizing required use of several existing large arrays, so we need to allocate memory and transfer it to the GPU

for the cascade classifier processing. We measured the execution time of the data transfer (including cudaMalloc and cudaMemcpy) to be a significant amount of time, as seen in figure 3.1.

Choice of Machine

Our machine choice yielded reasonable results. The use of the GPU gave around the expected amount of speedup that we predicted, and the CPU multi-threaded program also produced a significant speedup over the sequential version. It is clear that the GPU implementation of window parallelization yielded much more speedup than the CPU implementation, which is expected given the varying number of threads than can be used in each architecture. GPU's has significantly more cores than CPU's, allowing the window-wise parallelization much more effective for bigger image sizes. However, the drawback of using a GPU machine is that data transfers between the host and the device memory is extremely heavy. In contrast, parallelizing on a CPU doesn't require any memory transfers since the same data from the master process can be accessed by the spawned threads. When we compare the overall execution time of the GPU implementation with the parallel CPU implementation, they are quite similar due to the heavy overhead of GPU memory transfers. This leads us to conclude that depending on the input image and the GPU machine, there are different trade-offs to consider when making a choice of which architecture to use.

Project Code Link

https://github.com/minji7608/viola-jones-parallel

References

Figures

Figure 1.1: https://docs.opencv.org/2.4/doc/tutorials/imgproc/pyramids/pyramids.html

Figure 1.2: https://en.wikipedia.org/wiki/Summed-area_table

Figure 1.3: https://www.researchgate.net/figure/Cascade-classifier-illustration-2_fig2_323057610

Sequential Starter Code

https://github.com/kernel-bz/machine-learning/tree/master/CML/Face/viola_jones

Additional Readings

https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf

https://www.vocal.com/video/face-detection-using-viola-jones-algorithm/

https://www.cs.ubc.ca/~lowe/425/slides/13-ViolaJones.pdf

https://sites.google.com/site/5kk73gpu2012/assignment/viola-jones-face-detection

Division of Work

Equal work was performed by both project members.