Parallelized String Art

Catherine Tianhong Yu: tianhony@andrew.cmu.edu
Nanxi Li: nanxil1@andrew.cmu.edu

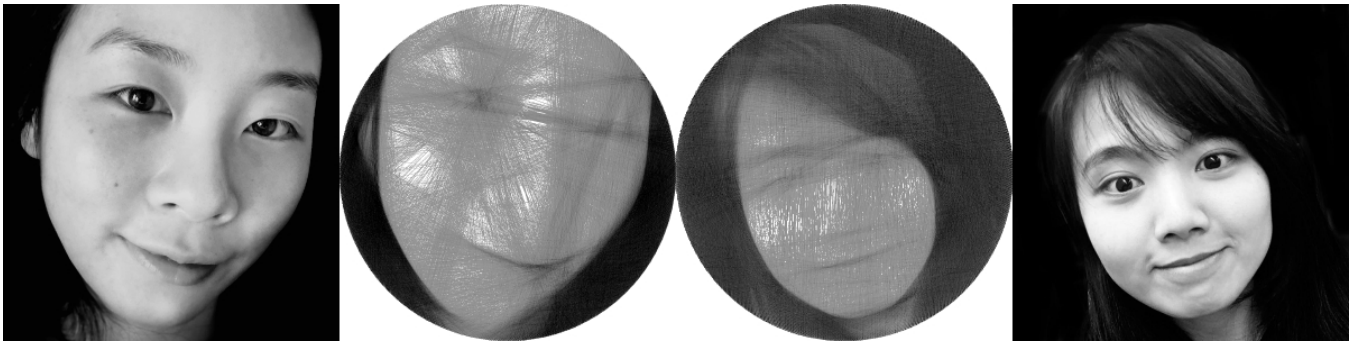

Figure 1: Example inputs and outputs with output diameter 1024px and 512 pins. Output on the left took 6171.59s to compute, and the output on the right took 7498.95s to compute.

1 SUMMARY

String art is an image solely composed of strings between pins around a circular canvas. We implemented a parallelized string art solver in C++ and CUDA that computes the string art best resembling the input image. We developed our algorithm from scratch based on the sequential greedy approach proposed in paper by Brisak et al[1]. We modified the proposed algorithm while implementing our sequential version of the solver, so that algorithm would have more parallelism to exploit while outputting more accurate string art image. We then developed our parallel version of the solver, which produces the same output as the sequential solver in a considerably shorter runtime. We were able to achieve an over 221x speedup on a 512*512 image with 128 pins.

2 BACKGROUND

First of all, what is string art? String art is a technique for the creation of visual artwork where images emerge from a set of strings that are spanned between pins[1].

Let P be the number of pins that spans the edge of the circular canvas evenly, and As we researched for solutions to this problem, there were 2 main types of solutions, both of which assume opaque threads:

- **Greedy Approach**: Starting from a pin p_0, the algorithm finds the pin $p_1, p_1 \neq p_0$, that the string between the 2 pins best fits the given image. Then starting from p_1, the algorithm finds the pin $p_2, p_2 \neq p_0$ that best fits the given image, until there is no new connecting between 2 pins can make the fit better, or a maximum number of strings reached. Note, the algorithm finds strings that are continuous (i.e, a string can only start from a pin where the previous string ends).

- **Modified Greedy Approach**: The modified greedy approach compensates for the fact that the greedy approach has the disadvantage of making decision of adding a string at an early stage that will later turn out to be a bad choice. In our case, an addition of one edge might bring the biggest benefit when it is chosen, but can then prevent a better solution later on. Thus, Brisak et al. proposed to further improve the results by an iterative removal of edges. (Figure 3). In particular, when the initial addition stage terminates, the modified greedy approach algorithm iteratively choose a string to test if removing that string will result in a better fit. If the removal of the string will result in a better fit, the string will be removed. The algorithm continues to search for strings whose removal can result in a better fit until no better fit from removal can be found. The algorithm then starts a second round of string adding. The algorithm alternates between the addition stage and the removal stage, until it is not possible to further improve the norm and the algorithm terminates. In addition, this algorithm further improves the quality of output by allowing strings to start from an arbitrary pin.

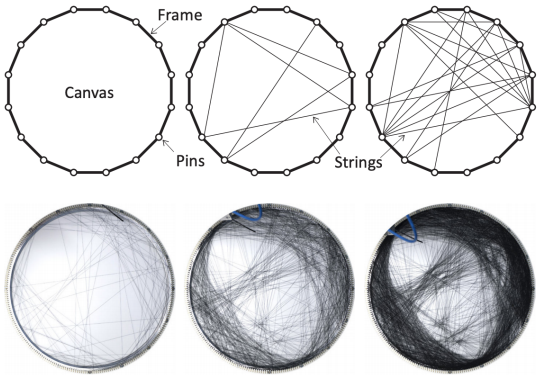

Figure 2: Construction of string art.[1]

Another algorithm is then applied to make the found pin pairs into a fabricable string art image. Since fabricability is not the goal of this project, we will not include this part of the algorithm.

while *true* **do**
  $j = \arg\min_i \|WF(x \pm e_i) - Wy\|^2$
  $\tilde{f} = \left\|WF\left(x \pm e_j\right) - Wy\right\|^2$
  if $\tilde{f} < f$ **then**
    $x = x \pm e_j$
    $f = \tilde{f}$
  else
    break
  end
end

Algorithm 1: Our algorithm computes a subset of all possible strings which are used together to reassemble the input image. Please note that this code summarizes the edge addition $(x + e_i)$ and edge removal $(x - e_i)$ operations of the algorithm as described in Section 5. Note that the vector e_i refers to the i-th column of the identity matrix.

Figure 3: Screenshot of the algorithm taken from[1].

For the Greedy approach, when looking for line to add, it checks at max P pin pairs(fixed starting pin). For the modified greedy approach, when adding, it checks for at max P^2 pin pairs, and when removing, it checks all the added pin pairs, which is bounded by P^2. We chose to use the **Modified Greedy Approach** because the output resembles the input better. However, since the problem is more complex, it is more challenging to implement a solver based on this algorithm.

We removed the assumption for completely opaque strings in our implementation. Our implementation allows overlay of strings. In other words, the cross point of 2 strings would appear to be darker than the strings themselves. This results in better resemblance at low resolution, which is crucial since high resolution images would take too long to be processed by the sequential solver. More importantly, this change allows us to exploit the string-level parallelism in the removal stage (refer to section 3 for more details), which contributes to the overall speedup massively.

Figure 4: Left: small resolution output on problem size $D = 256, P = 128$ using opaque string as assumed in the algorithm in Figure 3; Middle: input image with pins graphed; Right: same problem size as the Left but using our approach after removing the assumption of completely opaque strings

3 OUR WORK FLOW

3.1 Overall Work Flow

Our work flow is illustrated in Figure 5) and our solver is shown as the pink box in Figure 5).
The input to our framework is a colored image(for benchmarking purpose, we used grayscale images for easier contrast tuning). We convert it into a integer number representation in the range of [0, 255] and denote it further as the column vector:

$$y \in [0, 255]^m \subset \mathbb{Z}^m \tag{1}$$

where m is the number of all pixels, concatenated row-wise. The output of the optimization algorithm is a binary array, where each entry corresponds to an edge which could be drawn on the canvas. Activated bins reflect edges which need to be drawn. We denote the output as the column vector

$$x \in \mathbb{B}^n \tag{2}$$

where n is the number of all possible string edges. It has the dimensional of $P(P-1)$.
The goal of the optimization problem is to determine the best way to to define a mapping F from the space of edges to the space of pixels, i.e.,

$$F : \mathbb{B}^n \rightarrow [0, 255]^m \text{ with } x \mapsto F(x) \tag{3}$$

and to determine the values of the elements of the vector x such that it delivers the best approximation of the input image. We cast the problem into a binary non-linear least squares optimization task:

$$\min_x \|F(x) - y\|^2 \text{ s.t. } x \in \mathbb{B} \tag{4}$$

And the optimization is solved using the algorithm in Figure 3.

3.2 Parallelism Analysis in Solver

Figure 6 illustrates the workload that we parallelized. Like mentioned above, there are at max P^2 pin pair combinations to check for when adding and removing a string. For each pin pair, we compute the L2 norm between:

- constructed image with the string edge specified by the pin pair added/removed
- original image

Addition Stage: Computing the L2 norm for each pin pair while adding the line is trivial. The L2 norm after adding a line can be calculated in parallel without any data dependency, because they are all reading the same original image and constructed image and makes no alteration to the images. All of the threads only computes what the L2 norm would be if all pixels representing the string between 2 pins is covered by an additional string.

Removal Stage: If we assume the lines are completely opaque, it is impossible to exploit string-level parallelism in the removal stage with a limited memory footprint. The reason is that when considering removal of a string, naively coloring all pixels representing the string white results in very inaccurate L2 norm calculation: if the pixel is also in the path of another string, the pixel should still be black while the string is removed. The only way to parallelize

strings in the removal stage is to re-construct the image completely with the string removed. This approach required each thread storing its own copy of the re-constructed image. Such memory footprint requirement is not achievable given that the problem at max has P^2 pin pairs.

Hence, instead of assuming the lines to be opaque, we allows overlapping of strings. The cross point of two strings appears to be darker than the rest of the strings that are not overlapped. In this way, when calculate L2 norm with a string removed, the algorithm computes the L2 norm with all pixels representing the string between 2 pins being a shade lighter. This way, the expected L2 norm after a removal is an approximate value very close to the L2 norm value calculated from re-constructing the image. The algorithm corrects this small approximation by re-constructing the image after the decision of removing a string is being made.

Solver Implementation Details: Let D be the diameter of the output image. To compute the L2 norm with one line added/removed, we first need to find the pixels that represent the line between the 2 pins. This can be done in $O(D)$ by simply calculating the slope between the 2 pins. Then we need to look at every single pixel in $O(D^2)$ to calculate the norm. Each subproblem has complexity $O(D^2)$ which is costly even when the image is not huge. We understand there is also pixel-level parallelism in this subproblem. However, with the limitation of memory footprint, it will exceed the hardware limit of CUDA if we were to exploit pixel-level parallelism while exploiting string-level parallelism.

In other words, we parallelized finding the *min* to perform *argmin* on Figure 3. All other computations in a iteration of the `while` loop can be done in constant time, but we still have the dependency between each iteration of the while loop.

4 APPROACH

We could not find any C/C++ implementation but we had reference to matlab [2] Java [3] implementations, in addition to the psuedocode in the paper[1]. We also did not want to use any libraries other than file I/Os to enable the most flexibility in parallelization it using cuda.
In the rest of the section, we will first introduce the approaches of the sequential and Cuda versions that we benchmarked on, and then discuss what did and did not work.

4.1 Sequential Version

Image is stored in a 1D `char*` array of size $D \times D$, concatenated by rows.
Pin pairs found are stores in 2 separate `int` arrays of size $P(P-1)$. Which one of the pin gets stores in whichever array does not matter, but elements at the same index in the 2 arrays form a pin pair.

Finding a line to add: For each pin pair, we copy the constructed image(constructed by chosen lines) and try 'adding' the line by increasing the pixel value of the constructed image. Then compute L2, and update the best pin pair and norm found as pin pairs are iterated.

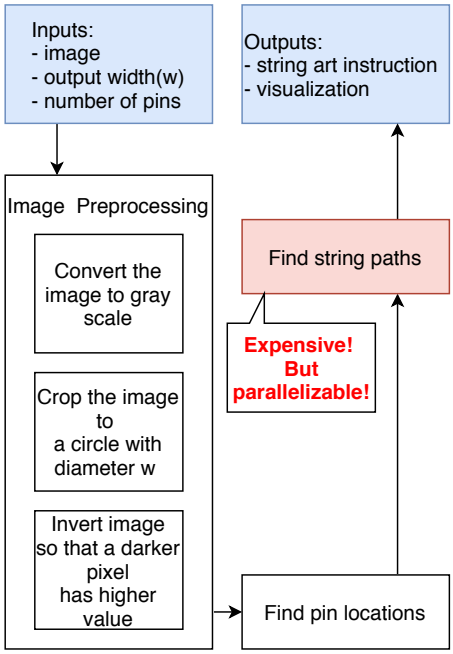

Figure 5: System flow diagram for the string art problem.

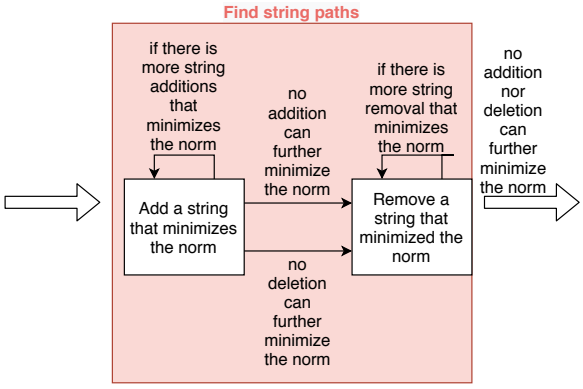

Figure 6: Our algorithm finds the optimal solution to string paths

Finding a line to remove: For each chosen line, we copy the constructed image(constructed by chosen lines) and try 'removing' the line by decreasing the pixel value of the constructed image. Then compute L2, and if the new norm is better the current norm, remove the pin pair and update current norm to new norm.

4.2 CUDA Version & Mapping to NVIDIA GPUS

The data structures used in CUDA are exactly the same. We purposefully modified our sequential version to make the the two implementations to use the same data structures and produce the same outputs.

Finding a line to add: `BlockDim(16, 16)`, `gridDim((P+15)/16, (P+15)/16)` so **each block has 256 threads, and each thread has a 'unique' pin pair combination, and vice versa each pin pair combination has a unique thread**. Note here that (p_0, p_1) and (p_1, p_0) refer to the same pin pair, so they each will receive a thread, but they will be doing the exact same computations. We are doing the same work twice, but since all computation is done in parallel, this would only increase the runtime marginally(in terms of overhead of launching the threads). The ease of implementation for indexing justifies the marginal increase overhead here. Each thread receives original image data and constructed image data from the host using `cudaMemcpy`, compute the expected L2 norm with the line "added", and returns the norm found back to the host using `cudaMemcpy`. The host then decides which line to add and re-construct the image.

Finding a line to remove: Let L be the length of pin pair arrays, i.e. the number of pin pairs found. `blockDim(256)`, `gridDim((L + 15)/16)`, so **each block has 256 threads, and each thread has a unique pin pair found, and vice versa, each pin pair found has a unique thread**. Each thread receives original image data and constructed image data from the host using `cudaMemcpy`, compute the expected L2 norm with the line "removed', and returns the norm found back to the host using `cudaMemcpy`. The host then decides which line to remove and re-construct the image.

This way, there is no data dependency between each thread, and thread is only accessing device data. Host loops through all norms returned and find the corresponding best one for adding/removing. We could further improve by using CUDA to reduce.

4.3 What did not Work

4.3.1 As-Is Algorithm Proposed in Paper. Like mentioned in our proposal, we anticipated the challenge of implementing a sequential version that contains all the data structures ready to be parallelized. Because our sequential version was extremely slow(Brisak et al.'s paper mentioned that some of the their solutions took 14+h to generate), we weren't able to see how our algorithm perform with larger inputs. The largest problem we were able to see output for on ghc machines without getting connection dropped(even with `nohup`) were $D = 512, P = 128$ which still took hours. Additionally, the image outputted is far from resembling the input images because the opaque string method proposed in paper only shows resemblance in high resolution result (Brisak et al.'s paper used $D = 4096, P = 512$).

4.3.2 Opaque Assumption. We implemented the parallel version to run a larger problem and further confirm the 'correctness' of our sequential code. In order to match the algorithm between sequential and parallel version, we have to develop parallel version as the same time as altering the sequential implementation. In order to parallelize the removal stage, we decided to change the opaque strings assumption proposed by the paper. The change to this assumption also allows us to output low resolution image with high resemblance to the input.

4.3.3 Queue as Data Structure. Our initial sequential code used a queue to store the strings that we have placed on the canvas. However, a queue is not a good data structure to exploit string-level parallelism since we want all strings stored in the queue to be accessed at the same time.

4.3.4 Pixel-Level Parallelism. In our 3.2 descriptions, we only had line level parallelism, but we also attempted pixel level parallelism in finding pixels associated with line and pixel shadings. We didn't get improvements from it because finding pixels associated with line is done for each line, so to parallelize it at a pixel level, we have to launch a kernel and allocate considerable amount of memory for storing pixel info. The memory consumption exceeded the hardware limit, so it is impossible to exploit pixel level parallelism expect for launch kernel in batches. The overhead of batching kernel launch is bigger than the speedup this parallelism can bring us.

4.4 What did work

4.4.1 Non-Opaque Strings. As mention in 3.2, we adapted non-opaque strings avoid redrawing the constructed image for every possible string removal. We also noticed that giving pixel values $\in \{0, 255\}$ to pixels not only makes the output looks not as pretty as desired, it also adds difficulties to L2 norm calculation since instead of being either black or white, each pixel value can be $\in \{0, 255\}$ when considering additional or removal. We used a more complex algorithm for calculating L2 norm, ensuring that the result is accurate. In this way, whenever we try to add/remove a line, we add/subtract a constant from the involved pixel, instead of redrawing the constructed image when checking for each removable line. This change not only made our CUDA implementation to have a great performance, it also made the output images MUCH MORE aesthetic.

4.4.2 Array as Data Structure. We used int array to store all types of data: all intermediate images, pixel information for strings, pin locations, L2 norms. This data structure allows us to access all entries stored at the same time. We used this fact to avoid any locks or barriers needed for our parallel solver, avoiding potential synchronization cost. With this data structure, we were able to combine a nested `while` loop(outer `while` loop is equivalent to the `while` loop in Figure 3, and the inner `while` loop loops through all found pin pairs) into one while loop where line removal has the same behavior as line addition: one line per iteration.

4.4.3 String-level Parallelism. We exploited all parallelism at string-level, both in the addition and removal stage. The more detailed analysis of parallelism is in 3.2. By exploiting all string-level parallelism, we were able to achieve an over 221x speedup on a $512 * 512$ image with 128 pins.

5 RESULTS

To evaluate our results, we ran benchmarking on GHC47-86 machines containing the NVIDIA GeForce RTX 2080 B GPUs. The baseline algorithm is designed for a single thread CPU and it was also ran on GHC47-86 machines. We measured clock times(in seconds) for computing the pin pairs needed to fabricate the string art(pink box in Figure 5). Clock times are used to analyze how our

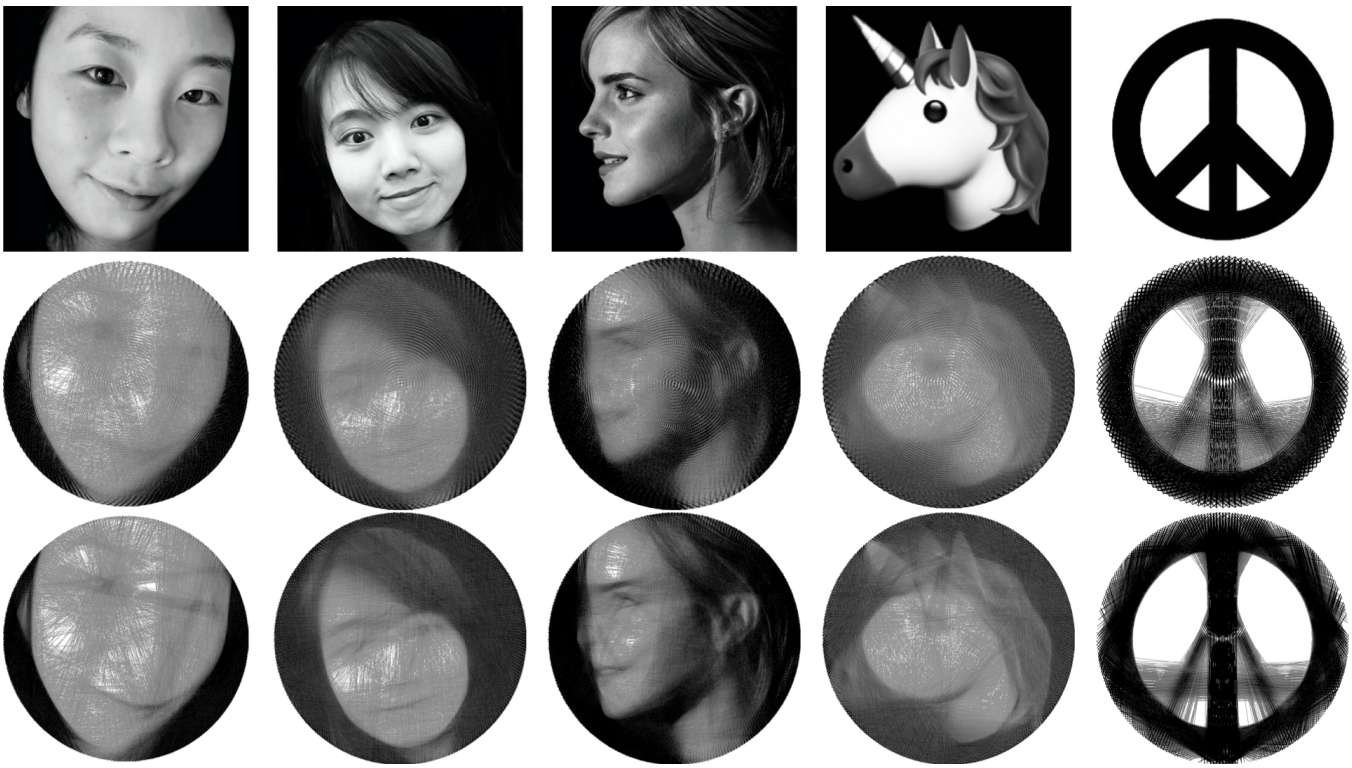

Figure 7: Example inputs and outputs using our system. Left to right image labels: catherine, nanxi, emma, unicorn, peace; First row: inputs; Second row: $D = 512, P = 128$ outputs; Last row: $D = 512, P = 256$ outputs. These are all outputted by our CUDA implementation. Notice given the same output image size, increasing the number of pins improves quality. Different contrast level are manually applied to these images, so some of the outputs appear darker/lighter than expected.

parallel implementation scales and compute the speedup to measure CUDA implementation performance.
Figure 7 displays the inputs(first row) we ran benchmarks on, and their outputs produced by CUDA implementation with $D = 512, P = 128$(second row) and $D = 512, P = 256$(last row). In the rest of the section, we will analyze the performance of our CUDA implementation by:

- comparing CUDA execution time with sequential execution time
- comparing CUDA execution time for fixed number of pins, but different image sizes
- comparing CUDA execution time for fixed image size, but different numbers of pin

5.1 Comparing CUDA execution time with sequential execution time

In Figure 8, when the problem size of $D = 512, P = 128$ is quite large, it takes 4 hours to compute the pin pairs sequentially, while our CUDA implementation can solve the same problems in 2 min with speedup of 72x-222x. Looking at these numbers, we are proud to say that GPU was definitely a good choice for our problem!

For a smaller problem size like in Figure 9, where the problem size is $D = 128, P = 64$. It takes 1.5min to compute the pin pairs sequentially, while our CUDA implementation can solve the same problems in 2s with speedup of 25x-42x. We can also see that the execution time varies quite a lot using the sequential version, but the variation is much smaller using CUDA. We speculate that this because this is because of the relatively small number of `while` loop iterations.

For a even smaller problem size in Figure 13, where the problem size is $D = 128, P = 32$. It takes 2.5s to compute the pin pairs sequentially, while our CUDA implementation can solve the same problems in 1s with speedup of 2x. We see that there is not much variance among images, and this is caused by when the image size and number of pins are small, different input images will yield the same output which causes similar execution time. However, even with small problems like this with `cudaMemcpy` overheads, we still see a bit speedup.

We can see that problem sizes are very important in our problem, and our CUDA implementation parallelism performance heavily depends on both image size and number of pins. Let's also take a look at execution times using cuda with a fixed parameter.

Image Name	catherine	peace	unicorn	nanxi	emma
CUDA time(s)	102.24	77.98	106.26	122.7	179.14
sequential time(s)	17883.5	17290.8	7644.04	19732.6	16735.6
speedup	174.92	221.73	71.94	160.82	93.42

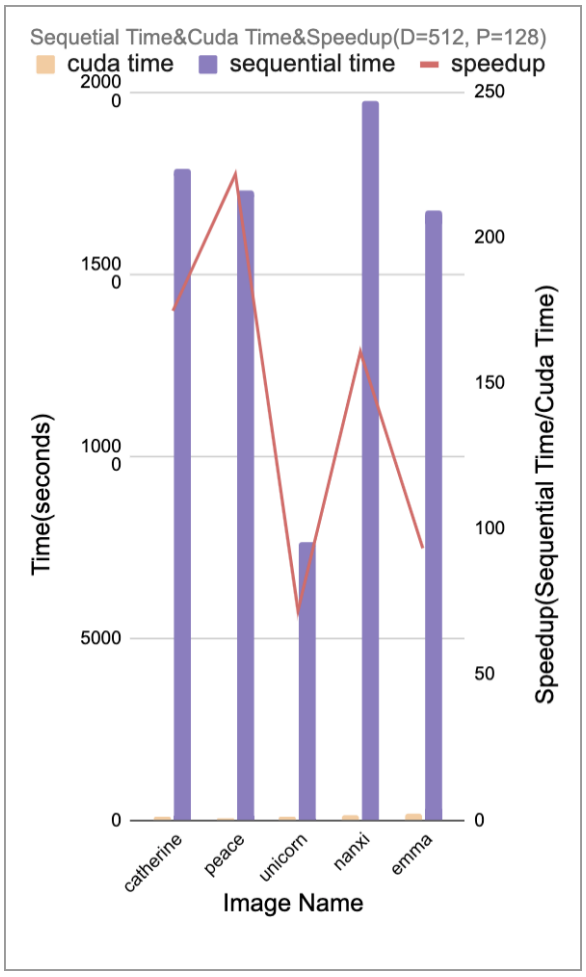

Figure 8: Comparing CUDA execution time with sequential execution time for $D = 512, P = 128$

5.2 Comparing CUDA execution time for fixed number of pins, but different image sizes

With number of pins fixed, as the diameter increases linearly, and the total number of pixels increases quadratically, our execution time is consistently heavily affected by the image size changing. Even though we don't have any pixel level parallelism, the distances between pin pairs increases as image size increases, and calculating the norm becomes more expensive. Therefore increasing image size increases the workload for each thread.

`while` loop. Furthermore, as image size increases, more pin pairs will be added to output pin pairs which adds to our sequential

Image Name	catherine	peace	unicorn	nanxi	emma
CUDA time(s)	2.01	2.67	2.03	2.91	3.02
sequential time(s)	51.06	95.9	86.75	57.25	94.67
speedup	25.40	35.92	42.73	19.67	31.35

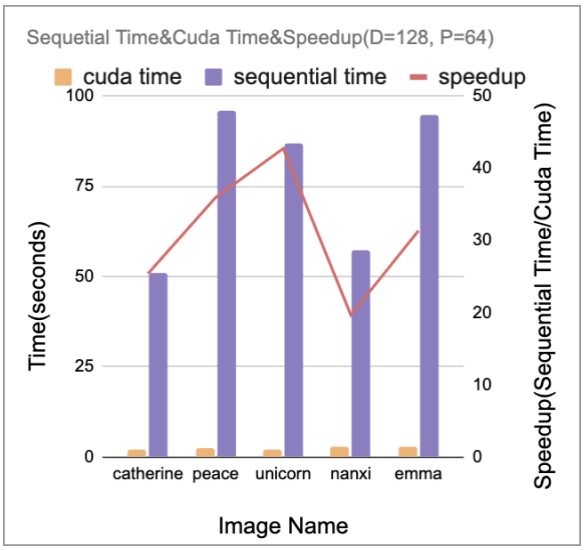

Figure 9: Comparing CUDA execution time with sequential execution time for $D = 128, P = 64$

Image Name	catherine	peace	unicorn	nanxi	emma
CUDA time(s)	1.14	1.12	1.12	1.11	1.14
sequential time(s)	2.54	2.61	2.57	2.59	2.61
speedup	2.23	2.33	2.295	2.33	2.29

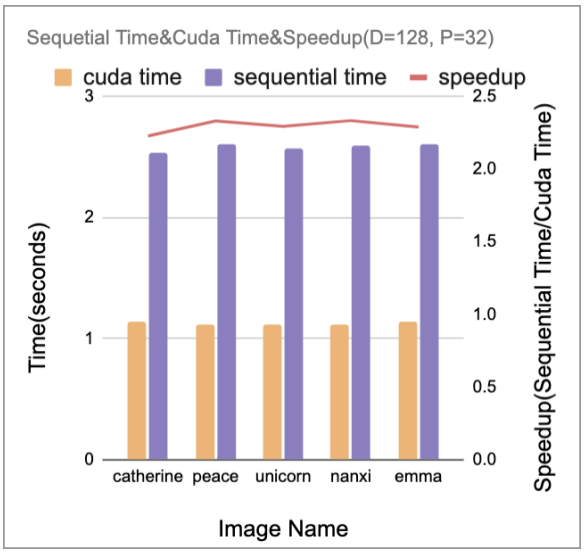

Figure 10: Comparing CUDA execution time with sequential execution time for $D = 128, P = 32$

5.3 Comparing CUDA execution time for fixed image size, but different numbers of pin

With the image size fixed, as the number of pins increases linearly, the number of pin pairs increases quarandically, our execution time is again consistently heavily affected by the image size changing. Although we have line level parallelism, we are again bottlenecked by the sequential `while` loop, because we now have more edges to add/remove, which means the number of while loop iterations drastically increases as well. We did not make measurements for these, but we speculate that the increase in while loop iterations contributed to the most of the differences here.

Image Name	catherine	peace	unicorn	nanxi	emma
D=128 P=128 time(s)	3.23	4.03	2.78	3.05	4.62
D=256 P=128 time(s)	16.48	17.58	15.05	16.75	24.77
D=512 P=128 time(s)	102.24	77.98	106.26	122.7	179.14

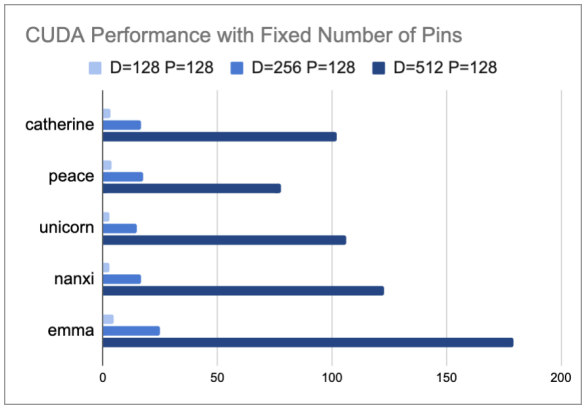

Figure 11: Comparing CUDA execution time with fixed number of pins $P = 128$ and different image sizes of $D \in \{128, 256, 512\}$

On another note, from analysis in Section 4.1, we believe that the larger the problem size, the better the speedup is! Though are not sure that if the problem size continues to increase, the increase in memory needed to store images and pin pairs will harm the performance. In Figure 1, problems of size $D = 1024, P = 512$ took 1h. Although we weren't able to measure the speedup because the sequential version will likely take days, we believe that the speedup still increases with such problem size.
To conclude the result section, we believe that GPU was a great choice as we have many independent sub-problems that are easily parallelized. Our parallelism performance is heavily problem size dependent, the larger the problem size is, the better the speedup is. And the sequential `while` loop limits the speedup.

6 CREDIT

The work for this project was evenly distributed between the two of us. We both did our own research for the problem and always discussed the design of our algorithm before one of us took the lead on the implementation. Catherine led the sequential implementation, and Nanxi led the parallel implementation based on our availability for the duration of the project. We also distributed the work for benchmarking and writeups.

Image Name	catherine	peace	unicorn	nanxi	emma
D=512 P=64 time(s)	21.46	21.81	21.82	21.93	21.84
D=512 P=128 time(s)	102.24	77.98	106.26	122.7	179.14
D=512 P=256 time(s)	298.08	578.63	308.9	340.03	471.63

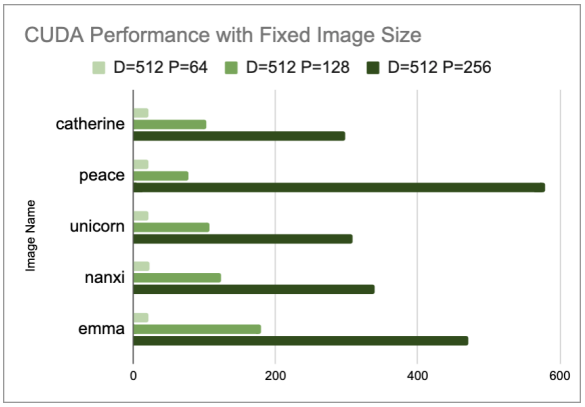

Figure 12: Comparing CUDA execution time with fixed image size $P = D = 512$ and different number of pins $P \in \{64, 128, 256\}$

Figure 13: $D = 512, P = 256$ outputs for Professor Mowry and Professor Railing using our CUDA implementation

REFERENCES

[1] Michael Birsak et al. "String art: towards computational fabrication of string images". In: *Computer Graphics Forum*. Vol. 37. 2. Wiley Online Library. 2018, pp. 263–274.
[2] Exception1984. *Exception1984/StringArt*. URL: https://github.com/Exception1984/StringArt.
[3] Jblezoray. *jblezoray/stringart*. URL: https://github.com/jblezoray/stringart.