15-418 Final Project Report

Ariel Davis (azdavis), Jerry Yu (jerryy)

May 7, 2018

Repo: `https://github.com/azdavis/parallel-portrait-mode`

1 Overview

We implemented portrait mode in parallel. Portrait mode has traditionally been done by high end DSLR camera hardware, in which the foreground is in focus and the background is blurred. We have chosen to use a software-only image processing approach to this problem, segmenting the image into foreground and background, and blurring only the background.

On our largest images, we were able to achieve upwards of 150x speedup with CUDA on GPUs, 11x speedup with OpenMP on CPUs, and 11x speedup with ISPC on CPUs..

Figure 1: Input Image

Figure 2: Output Portrait Mode Image

2 Algorithm

We used a simplified version of the grab-cut algorithm, with the idea of getting the color distribution of the background to find the foreground. Our algorithm takes in an input image and returns the image in portrait mode.

Each step of the algorithm will be referred to its shortened name in parenthesis.

1. Designate the borders of the image as the background. Top, left, and right $\frac{1}{8}$ of image.
2. Get a color distribution of the background region. For each background pixel, increment the appropriate color count bucket. (Color Counts)
3. Create a foreground mask by finding every pixel with a color not in the background color distribution. Filter background colors that make up less than 1 percent of the background area. (Build Mask)
4. For every unmasked pixel, add it to the mask if two of its neighboring pixels were part of the mask. (Refine Mask)
5. Blur all the pixels that are not within the foreground mask. We used a convolution with a circular filter to emulate the bokeh effect. (Blur)
6. For each pixel in the mask, add the original pixel of the image into the blurred version of the image.

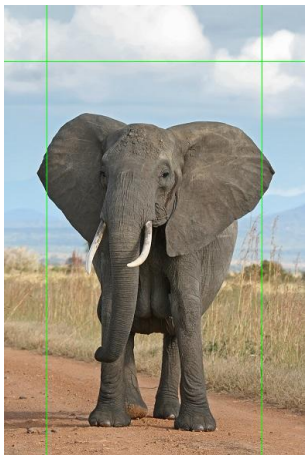


Figure 3: Background Region (Step 1)

Figure 4: Foreground Mask (Step 5)

2.1 Data Structures

Each image is represented as a struct with the width, height, and an array of pixels. Each pixel contains a red, green, and blue value. The size of the array is the number of pixels in the image.

We stored the color counts by creating color buckets for ranges of RGB values. Essentially, all colors in one bucket would be within 15 red, green, and blue value. This results in $\left(\frac{255}{15}\right)^3$ different buckets. So, we are able to store the color counts with an array, with each element representing the count for that bucket.

The mask is simply an array of chars the size of the number of pixels in the image. If the value in the mask is 1, the corresponding pixel is in the foreground. Otherwise, it is in the background. We chose chars to minimize the size of the data structure.

2.2 Background

The blur is the most computationally intensive part of the algorithm. Each pixel of the blurred image needs to compute the average RGB values for a circle of pixels around it. However, the blur is data parallel, as each pixel is independent.

However, each step of the algorithm has to be finished before the next step. This requires synchronization after each step.

3 Approach

We were interested in comparing the GPU and CPU implementations of our algorithm, so we implemented it in CUDA and OpenMP. To develop and solidify our algorithm, we first wrote an implementation with Python's NumPy. We experimented with several edge detection, gradient, and snake methods and decided the color distribution method gave us the best accuracy for foreground detection.

We then implemented the sequential C++ before parallelism. To our surprise, the sequential C++ version was significantly faster than the unoptimized Python. The Python program took 51 minutes for our smallest elephant image with a blur filter size of 50px, and our C++ version took 2.5 seconds for the same results.

4 CUDA on GPU

We utilized the NVIDIA 1080x GPUs on the GHC machines with CUDA. The general strategy for CUDA was to parallelize across the pixels in the image to take advantage of the high number of ALUs and threads.

4.1 Color Counts

To get the color counts, we assigned each thread to each pixel. Each thread then did an atomic add to appropriate bucket in color counts. An atomic add was needed because multiple pixels

could simultaneously have the same color.

We also tried assigning one thread to one color to iterate through the image, but that was repeating the sequential algorithm of looking at each pixel.

Another implementation was to create copies of the color counts data structure for each thread to process, but one of our timing bottlenecks was performing cudaMallocs to initialize memory in CUDA. (discussed later on)

So, we moved forward with the atomic strategy. We initially thought the contention all the threads would slow this part of the algorithm, but our timing showed us atomic allowed this step to be performed in almost 0 time.

4.2 Build Mask and Refine Mask

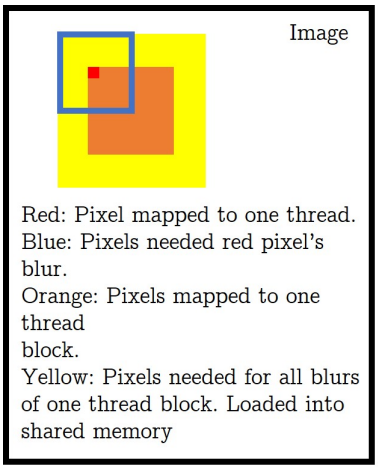
The CUDA implementation of build mask and refine mask were more straight forward. Each thread was simply mapped to one pixel.

We tried to compute both steps within one launch, but realized we needed to synchronize or communicate between warps. The overhead of another launch was less, so we just kept the two steps in two launches.

4.3 Blur

We were able to take advantage of the data parallelism for the blur, as each pixel's computation was independent from those of other pixels. So, each pixel is mapped to one CUDA thread. Additionally, pixels spatially close are mapped to the same block. This allowed us to take advantage of the high amount of threads in each SM. This mapping significantly decreased the arithmetic computation per thread, but we realized that we had a memory bottleneck.

We noticed that the pixels close together needed to access the same neighboring pixels to perform the convolution. So, we loaded every pixel needed for that block's blur into shared memory. The kernel that contained the weights for how much each neighboring pixel contributed to the average was also loaded into shared memory. By having each thread load data into shared memory, the cost of loading it is very low while allowing every memory access to be from shared memory.



Thread mapping and shared memory for CUDA blur

5 OpenMP on CPU

We utilized the 16 thread CPUs on the GHC machines with OpenMP. For CPUs, there was not enough threads to parallelize across every pixel like on GPUs. So, we chose to parallelize across rows of the image, keeping arithmetic intensity high while taking advantage of parallelism.

5.1 Color Counts

We chose not to parallelize the color counts step in our algorithm. From our profiling, getting the color counts made up less than 0.03 percent of our time, and that overhead with parallelism would not increase our speedup.

We first tried the atomic strategy we used for CUDA. Each thread was assigned a row and performed an atomic add to a color bucket for each background pixel. However, the overhead from gaining exclusive control to the bucket made the implementation around 2x slower than sequential. For large elephant, sequential was around 0.038s and with atomic parallel it was 0.95s.

Additional methods would be to create local copies of the data structures for each thread and combining them in parallel. But this has the disadvantage of requiring much more memory allocation, as well as the cost of the combining step.

5.2 Build Mask and Refine Mask

We parallelized building and refining the mask by parallelizing over the rows of the image. We chose rows because it gave us enough arithmetic computation while still giving us the

granularity we needed for the computation.

We experimented with using SIMD in the inner loop over each column, but that was around 1.5x slower than without SIMD. This is probably because there is an uneven workload in one row. Each loop iteration only writes to an array if there is a conditional, so there is uneven work that causes SIMD to take longer.

5.3 Blur

Similar to the previous steps, we parallelized over the rows for the blur. However, the work of each row varied significantly. This is because rows with pixels in the foreground are not blurred, saving 50^2 computations with a filter size of 50. So, we used dynamic scheduling with OpenMP in order to distribute the work evenly. With parallelizing over rows and dynamic scheduling, we were able to maintain a small granularity while still having even work among the threads. With dynamic over static scheduling we were able to achieve around 50 percent speedup for blur on large images. (From 15s to 10s)

We also tried adjusting the chunk size with dynamic scheduling in order to reduce the amount of overhead with scheduling the work. However, changing the chunk size to 5 or 10 did not increase the speed. Increasing chunk size only increased the time differences between threads. We also experimented with using SIMD over the columns for the inner loop of the blur. However, this resulted in a blur that was around the same time as without SIMD. This is because new colors for each pixel were not adjacent in memory and required scatter operations to write to them. Many of the memory accesses also required gather operations, which slowed down the vectorization speedups from SIMD.

5.4 Blur with ISPC

We wanted to explore further possibilities with CPU parallelism, so we implemented the blur in parallel with ISPC. Similarly to our strategy with OMP, we we chose a dynamic scheduling method, creating a task for each row and launching them across the cores. We also utilized SIMD in order to process each column. It was interesting to explore the implementation details between ISPC and OMP with almost the same algorithm.

We attempted to implement more parts of our algorithm with ISPC to compare, but we encountered some bugs in the compiler. Nevertheless, the blur was the most computationally intensive part of our algorithm. So, we used a hybrid approach with ISPC for the blur and OpenMP parallelism for the rest of the program.

6 Results

6.1 Total

Below are the times and speedup we achieved for each method. Each of the times are taken from the minimum of 10 times we ran the program. We chose the minimum of the 10 trials in order to get as close as we could to how long the program actually took. The speedup is the time compared with optimized single-threaded C++ version.

Image	Size	C++	OMP	ISPC	CUDA
Elephant	389×584	2.0887s	0.1930s	0.2170s	0.1409s
Bluejay	1200×832	11.8096s	1.0632s	1.2070s	0.2034s
Tiger	1400×845	14.6917s	1.3536s	1.5463s	0.1950s
Flower	2242×2112	72.0560s	6.0286s	6.1216s	0.3859s
Purp	2944×2609	88.9941s	7.5728s	7.6757s	0.6067s
Large Elephant	2592×3888	113.2247s	9.8219s	10.7315s	0.7369s

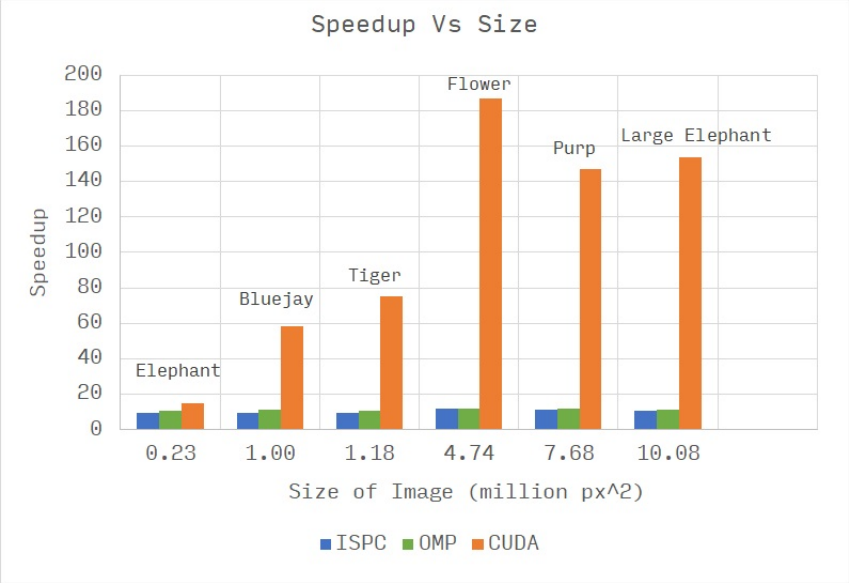
Image	Size	OMP Speedup	ISPC Speedup	CUDA Speedup
Elephant	389×584	10.8240x	9.6238x	14.8264x
Bluejay	1200×832	11.1078x	9.7846x	58.0582x
Tiger	1400×845	10.8540x	9.5010x	75.3517x
Flower	2242×2112	11.9523x	11.7707x	186.7298x
Purp	2944×2609	11.7518x	11.5943x	146.6925x
Large Elephant	2592×3888	10.7315x	10.5507x	153.6602x

Overall, we were satisfied with the speedup we achieved. With the CPUS having 16 hyperthreads, we were able to achieve 11x speedup with OMP and ISPC. With GPUs, we achieved 150x speedup over the CPU sequential implementation, reducing operations that took minutes to less than a second.

It was also very interesting comparing the performance of many different platforms and the changes to implementations we had to make for OMP, ISPC and CUDA.

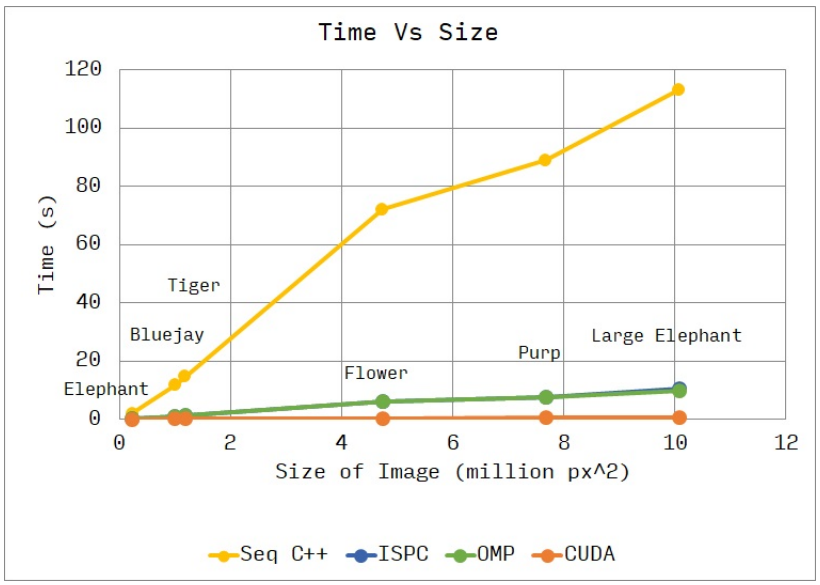
We can see that the GPU implementation easily beats out both CPU implementations. This is mostly due to the hardware nature of GPUs, having enough threads to be able to parallelize operations per pixel. When we compare a 20 SM NVIDA GTX 1080 handling thousands of threads each with a CPU that has 16 threads, it is easy to see the sheer power of GPUs. GPUs hold true to their name as graphical processing units, as they are the ideal hardware to perform image operations.

The fundamental difference between are GPU and CPU implementations was that with the number of threads available, we had to parallelize over rows in CPUs while we were able to parallelize over pixels in GPUs. We also took advantage of different concepts in parallel programming on each platform. For CPUs, we explicitly used dynamic scheduling and SIMD. For GPUs, we used shared memory with thread blocks.



It is also interesting to see the relationship between the size of the image and the speedup achieved. As the image is larger, the arithmetic intensity increases. However, the nature of the image itself can also contribute to speedup. Because the flower image had a larger region of background, more blur was needed and the GPU was able to achieve more speedup than larger images like Purp.

In addition, we found out that speedup with CPUs was relatively constant for each differently sized image, hovering at around 11x. Meanwhile, speedup on GPUs heavily fluctuated and significantly increased with image size. This is likely due to the higher overhead of launching CUDA threads versus the overhead of using threads.



From each timing, we saw an almost linear relationship between time and size of image for each implementation. Interestingly, we also saw that taking longer to compute did not necessarily mean parallelism was more effective, as we achieved the most speedup with

Flower.

6.2 Algorithm Speedup

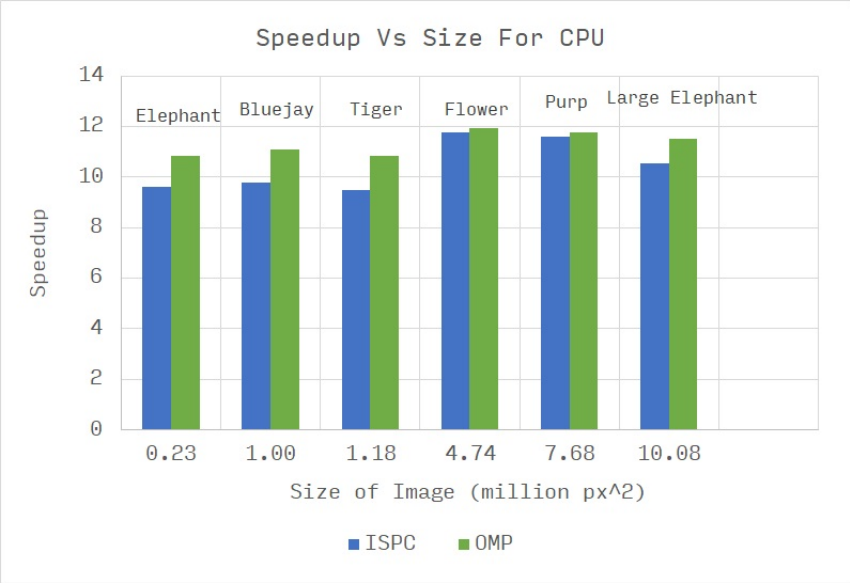
Below are the specific speedups for each step of our algorithm. We measure them from our largest image Large Elephant, which is 2592×3888 px.

Item	C++	OMP	Speedup	ISPC	Speedup	CUDA	Speedup
init	0.0118	0.0087	1.3505	0.0144	0.8203	0.1803	0.0655
color counts	0.0283	0.0181	1.5624	0.0285	0.9946	0.0000	726.2308
build mask	0.0640	0.0152	4.2228	0.0148	4.3311	0.0026	24.8173
refine mask	0.0337	0.0162	2.0788	0.0183	1.8463	0.0000	1984.2353
blur	112.7321	9.4200	11.9673	10.3574	10.8842	0.1695	665.0706
clean up	0.3547	0.3436	1.0325	0.2981	1.1898	0.3844	0.9228
total	113.2247	9.8219	11.5278	10.7315	10.5507	0.7369	153.6602

With almost every step of our algorithm, CUDA was able to achieve significant speedup over the sequential implementation. The largest amount of time spent for CUDA was the sequential writing the image to the file. CUDA was also unique because it had to transfer memory like the image from the CPU to the GPU. This led to the second most time spent in the init, which is due to the bandwidth of the interconnect. Because most of our time in CUDA was spent in file IO, we were satisfied with our implementation.

For the CPU implementations, the bottleneck was the blur operation. Even though we were able to achieve speedup in the blur step, it still took most of the time. We parallelized across the build mask and refine mask steps, but their computational intensity was not high enough to allow for significant speedup. Instead, of the overhead of starting the parallelism only lead to around 2 to 4x speedup.

6.3 ISPC vs OMP



It was very interesting to compare two different CPU parallel frameworks on the same code and same hardware. Both approaches used dynamic scheduling and small granularity to ensure equal work per worker.

Although ISPC utilized SIMD instructions in the blur process, using SIMD in ISPC did not increase speedup over OMP. We think this is mainly because the ISPC compiler told us we had several gather and scatter operations. One particular costly operation is when the member of a gang accessed each red color value from the pixel. Since there is also a green and blue value per pixel, the memory was not contiguous and needed a gather to read and scatter to write.

Additionally, our weights were stored as floats and the colors were stored as unsigned int8, and the ISPC compiler told us that doing this conversion was very costly. The OMP and ISPC implementations were both very close in time and speedup, but the OMP turned out to be slightly better at the end.

7 Conclusion

Overall, we were able to apply what we learned from 15-418 to speed up an image processing application. By developing our application in three different parallel frameworks, we learned a lot about the differences in implementation between each framework and their effects on speedup. Each platform had their own advantages and disadvantages, mainly rooted in hardware like the number of threads or the cost of data transfer.

In the future, we hope to be able to figure out how to further vectorize our CPU code to achieve speedup with SIMD. For example, we could change the our image data structure to

ensure contiguous memory access. We also would like to implement our algorithm on parallel image processing frameworks like Halide.

8 Work Distribution

Ariel and Jerry both performed equal work. Ariel did many of the OpenMP optimizations, wrote the Makefiles, built the testing framework, and set up all of the different parallel environments. Jerry helped with the OpenMP and wrote the ISPC and CUDA optimizations. We both also developed the Python and C++ sequential implementations. We both discussed implementation ideas and strategies with each other, and collaborated on each report.

9 References

Our image segmentation algorithm was a simplified version of the grab-cut algorithm. `https://dl.acm.org/citation.cfm?id=1015720`

10 Additional Data

Time is in seconds.

10.1 Elephant (389×584 px)

Item	C++	OMP	Speedup	ISPC	Speedup	CUDA	Speedup
init	0.0004	0.0003	1.2633	0.0003	1.1485	0.1265	0.0030
color counts	0.0004	0.0003	1.1442	0.0003	1.2086	0.0000	11.4062
build mask	0.0011	0.0003	3.8408	0.0003	3.5691	0.0001	11.8085
refine mask	0.0007	0.0002	3.1698	0.0002	3.1111	0.0000	56.0000
blur	2.0756	0.1825	11.3708	0.2057	10.0882	0.0048	432.9497
clean up	0.0106	0.0093	1.1372	0.0101	1.0454	0.0094	1.1263
total	2.0887	0.1930	10.8240	0.2170	9.6238	0.1409	14.8264

10.2 Bluejay (1200 × 832 px)

Item	C++	OMP	Speedup	ISPC	Speedup	CUDA	Speedup
init	0.0012	0.0012	0.9839	0.0013	0.9420	0.1486	0.0082
color counts	0.0016	0.0015	1.0456	0.0016	0.9994	0.0000	44.5143
build mask	0.0046	0.0010	4.6623	0.0010	4.5157	0.0003	14.5016
refine mask	0.0030	0.0012	2.5984	0.0013	2.2870	0.0000	230.4615
blur	11.7677	1.0259	11.4706	1.1683	10.0721	0.0222	528.9811
clean up	0.0315	0.0324	0.9714	0.0334	0.9418	0.0322	0.9790
total	11.8096	1.0632	11.1078	1.2070	9.7846	0.2034	58.0582

Figure 5: Input Image

Figure 6: Output Portrait Mode Image

10.3 Tiger (1400 × 845 px)

Item	C++	OMP	Speedup	ISPC	Speedup	CUDA	Speedup
init	0.0014	0.0011	1.2745	0.0015	0.9403	0.1280	0.0110
color counts	0.0018	0.0017	1.0614	0.0025	0.7235	0.0000	56.1875
build mask	0.0057	0.0012	4.8380	0.0012	4.6639	0.0003	16.5814
refine mask	0.0040	0.0015	2.6534	0.0017	2.3668	0.0000	284.8571
blur	14.6382	1.3011	11.2503	1.5014	9.7496	0.0277	528.8571
clean up	0.0406	0.0469	0.8640	0.0380	1.0665	0.0389	1.0425
total	14.6917	1.3536	10.8540	1.5463	9.5010	0.1950	75.3517

Figure 7: Input Image

Figure 8: Output Portrait Mode Image

10.4 Flower (2242 × 2112 px)

Item	C++	OMP	Speedup	ISPC	Speedup	CUDA	Speedup
init	0.0036	0.0037	0.9586	0.0069	0.5133	0.1528	0.0233
color counts	0.0074	0.0068	1.0950	0.0115	0.6479	0.0000	185.8500
build mask	0.0193	0.0060	3.2106	0.0062	3.1106	0.0014	13.5492
refine mask	0.0111	0.0068	1.6398	0.0083	1.3459	0.0000	585.0526
blur	71.8723	5.8649	12.2547	5.9443	12.0910	0.0868	828.1178
clean up	0.1423	0.1405	1.0131	0.1445	0.9851	0.1448	0.9830
total	72.0560	6.0286	11.9523	6.1216	11.7707	0.3859	186.7298

Figure 9: Input Image

Figure 10: Output Portrait Mode Image

10.5 Purp (2944 × 2609 px)

Item	C++	OMP	Speedup	ISPC	Speedup	CUDA	Speedup
init	0.0075	0.0104	0.7245	0.0068	1.1082	0.1948	0.0385
color counts	0.0201	0.0171	1.1788	0.0093	2.1705	0.0000	410.5714
build mask	0.0424	0.0110	3.8613	0.0102	4.1756	0.0017	25.0018
refine mask	0.0203	0.0105	1.9321	0.0104	1.9458	0.0000	967.6190
blur	88.5988	7.2521	12.2169	7.4000	11.9729	0.1149	771.1352
clean up	0.3049	0.2717	1.1221	0.2391	1.2755	0.2952	1.0329
total	88.9941	7.5728	11.7518	7.6757	11.5943	0.6067	146.6925

Figure 11: Input Image

Figure 12: Output Portrait Mode Image