15-418 Final Project
2D Ray Tracer
By Emily Zhang and Rahul Khandelwal

Summary:

We implemented a parallel 2D ray tracer using CUDA, OpenMP, and ISPC to compare the performance of these implementations and optimize the speedup of our algorithm. Our results show that our best implementation achieves over 100x speedup from the sequential. By comparing speedups and execution time between algorithm implementations, we determine that reasonable video frame rates can be supported to render ray emitting light sources with a single 3000 core GPU (standard in gaming hardware).

Background:

Light is made up of electromagnetic waves that propagate through space, colliding with particles and dispersing in the process. Ray tracing aims to simulate this interaction via computations. We model light emanating from a light source as a large number of rays pointing in all directions. To compute the effects of this light source on an image, each computational step involves a single light ray from a source, updating its position by a delta value, checking the pixel on the screen which the ray currently resides in, and applying an appropriate score to that pixel to update its color based on the ray's distance and color information.

As a 2D ray tracer, the necessary components for our program are a set of light sources, a 2D image or environment to apply the light sources to, and knowledge of where the edges are that would block the light rays. We then output the image with the light effects applied. This can be seen in Figure 1 below:

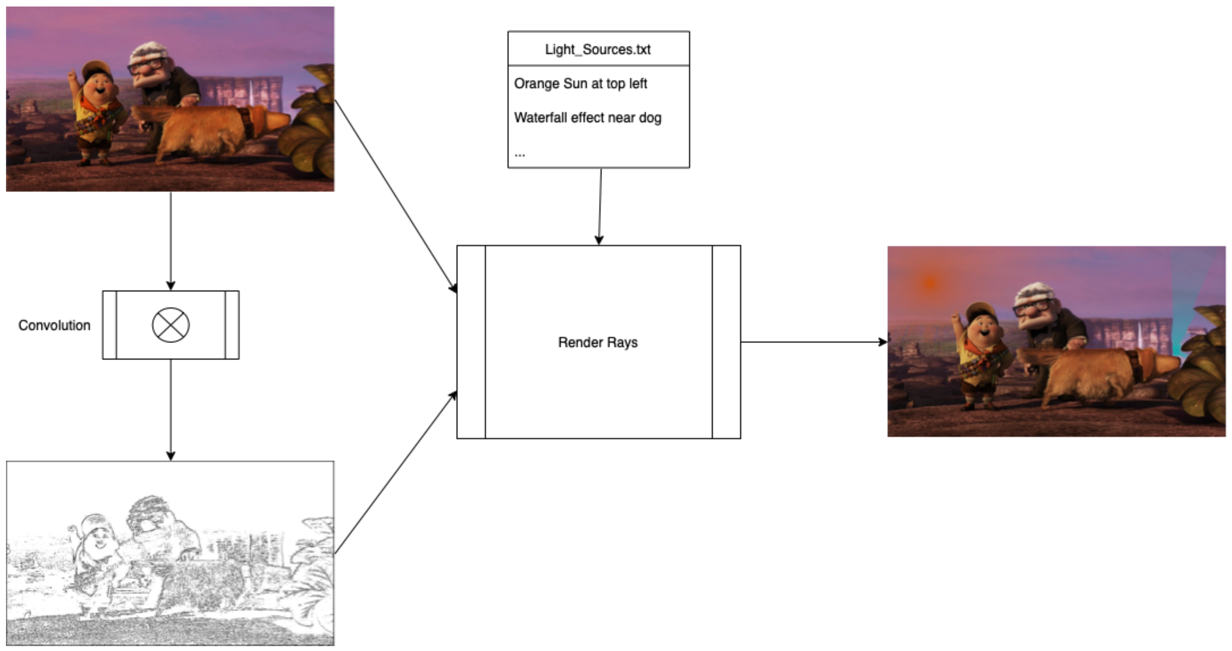


Figure 1

In order to gain knowledge of where edges are in the image, we precompute an edge detection convolution on the original image. This algorithm takes in a colored image as an input which is implicitly converted to grayscale using the png++ library, and outputs a black and white image such that white pixels represent non-edge space and black pixels represent edges.

We represent the light sources input as lines from a text file, denoting the location, intensity, and RGB color of a light. This information is stored in a lightray data structure that we created. We represent our image inputs as png-type objects from the library png++, which is a 2D-array-like structure that points to objects in different locations in memory that represent each pixel's information. Applying edge detection and ray tracing both require reading and writing pixel values to a png structure. In several instances, we also create 2D arrays with contiguous memory because not all compilers are compatible with the png++ library. We include the overhead of copying this data over in our time measurements.

The primary computational cost of image processing algorithms such as ray tracing lies in the large number of pixels that require calculations and updates. For a standard computer screen of 1920x1080, this is over two million pixels. A naive ray tracing implementation would require visiting each pixel up to [#rays] times, and [#rays] happens to be 5000 for our implementation of light sources. This is what only 12 rays emanating from a source within a single pixel would look like workload wise to a CPU (See Figure 2)

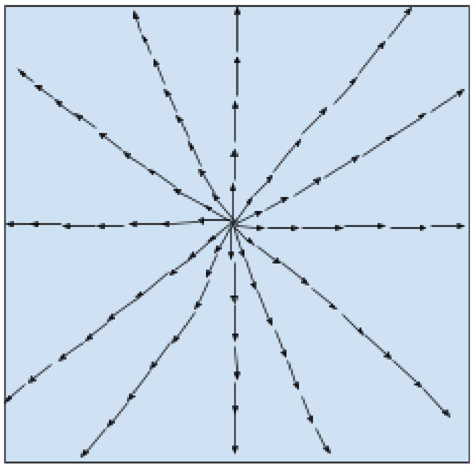


Figure 2

Every light source contributes to its own m*n matrix of "light scores" to tell each of the m*n pixels how much to update its color by. Each head of a vector in Figure 2, represents a step where the computer must increment the light score from a source at the resident pixel, not necessarily in order as our more complex algorithms will outline. Once all of the light source matrices have been computed, each of them must be reduced via averaging the rgb scores across the matrices at every index into one encompassing color matrix. This final color matrix is then applied as a single mask on the original input image. The result is shown as the sole output of the render rays algorithm in Figure 1.

Depending on the algorithm, there are many opportunities for parallelism. For example, if pixel computations are reasonably independent of one another, parallelization across pixels will be beneficial. Similarly, parallelization across light sources or even across rays are all valid opportunities for speedup. The convolution algorithm has a particularly good prospect for speedup via parallelization across pixels because each computation is independent of other pixels and because the depth of instruction is consistent across all pixels. The ray tracing algorithm yields less benefit from parallelizing across pixels. Assigning threads to individual pixels will certainly have load imbalances as some pixels channel exponentially more rays than others. The other option is to use an acceleration structure to decide appropriate spatial boundaries dynamically. Efficiently communicating possibly thousands of rays across these boundaries between threads would be a difficult time loss to minimize and may involve a rather convoluted implementation. Instead, parallelization across light sources and rays are promising to us because each computation is largely independent of one another, but possibly facing contention

Approach:

An overview of how the algorithm works was detailed in the background section. Here are some of the finer details that go into the different implementations, both parallel and sequential.

<u>Configuration</u>

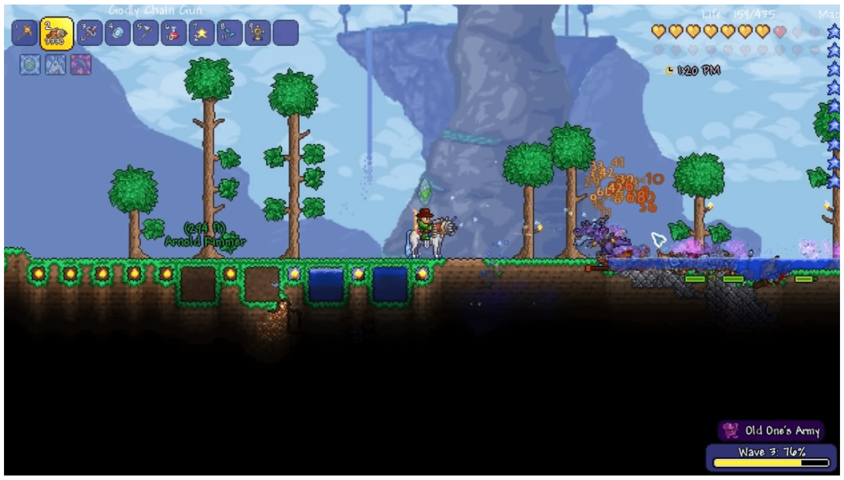
All of the implementations make use of or attempt to make use of the png++ library. Since we wanted to work with png files as they are a more modern file type, we figured this or some other library would aid our development. Though it was useful from a coding standpoint, configuring png++ to work on Mac and with OpenMP was not initially easy. Directly downloading the source files of the png library and linking the directory worked fine initially. The newer M2 Mac chips do not have support for OpenMP with their default clang compiler for C/C++. To combat this issue we made use of a homebrew solution for an x86 instruction-based compiler that could be run on M2. This had support for OpenMP but there were complications for having to reinstall png++ libraries and dependencies that worked with our new compiler. Likewise, for ISPC with tasks we had to download the ISPC compiler via homebrew for Mac. Other architectures which we ran code on such as the GHC and PSC machines were generally not too much of a hassle to set up our code on.

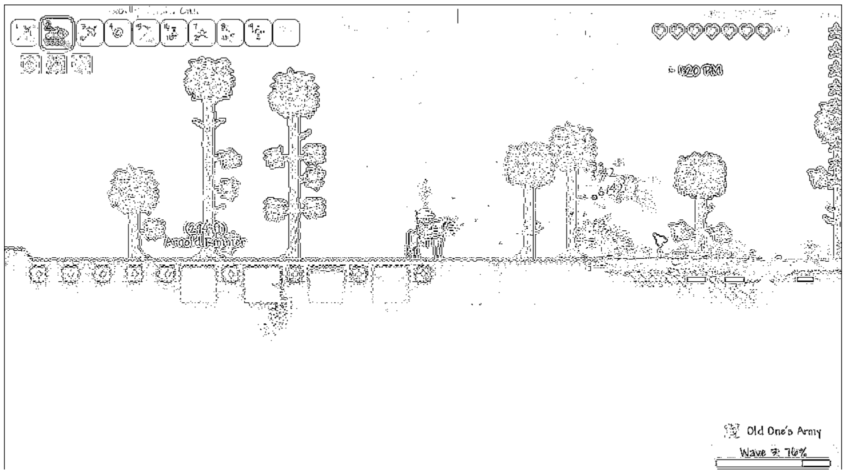
<u>Algorithm Details - Convolution</u>

The convolution step uses the edge detection kernel seen on the right. For any pixel, the mask is positioned such that its middle is aligned with the pixel. The mask represents weights, and the value of a pixel is determined by summing the mask weights multiplied by the value of pixels in a 3x3 area. The

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

result of applying this kernel is a mostly-black image with patches of gray and white denoting the edges that were found. For simplicity for the ray tracing algorithm, we adjusted the output such that the pixels were either fully white or fully black (i.e. value 0 or 255) while providing the best estimate of the edges as possible. We found that for standardly lit images, this threshold was around value=120, or ~50% brightness. We proceed with the assumption that we will be dealing with reasonably lit images. An example of this threshold applied to our benchmark image is below:





The convolution step is essentially a double for-loop that iterates across the rows and columns of the image to compute a value for every pixel. The computations for each pixel rely only on the base image and not on the computations of other pixels, so implementing parallelism across pixels via parallelism across for-loops with OpenMP was simple.

Another possibility for parallelism lies in the mask application process, for which the computation of the weight times the value of any particular pixel is independent of other pixels, but these results must be somehow combined with the computations of other pixels in the mask.

This can be done with a gather in parallelization models such as MPI, but the overhead of these models is so large that there is likely no benefit from using them.

Since the convolution pixel computations are independent of each other, and each computation undergoes the exact same process, we can expect maximum vector utilization for SIMD parallel processing methods. Thus, we thought that parallelizing convolution with ISPC would yield good speedup. ISPC is a compiler for a variant of the C language, which means that it does not have access to the same png parsing/processing libraries that were used for the sequential implementation of our algorithms. This requires that each image be converted to a 1D array containing all of the pixel values before passing into the ISPC functions, and that each image be converted back to a png image afterwards. Since convolution is something that happens only once for each image, we believe that this overhead should be included in our results (as opposed to a computation that requires opening the image once and then running multiple computations on the same image).

It should be noted that ISPC executes computations on a single core by default. A further optimization that we can do is to implement ISPC with tasks to achieve both multiple data parallelism and multicore parallelism. We wrote code to execute this by assigning each task a portion of the image to work on. Each task knows what rows they should work on via the taskIndex built-in parameter for ISPC. Our initial implementation split the image by rows, splitting it into chunks vertically. We then changed the implementation such that it split the image by columns, which may yield better load balancing since for our primary use case of applying ray tracing to a Terraria image, the upper half of the image is usually much lighter than the lower half of the image.

We were unfortunately unable to get ISPC with tasks to compile. After failing to find solutions online, we instead implement multi-core parallelism via OpenMP. Instead of using OpenMP to parallelize across individual pixels, we use OpenMP as a way to launch tasks via the same manner described above.

Algorithm Details - Parallel Ray Tracer

As mentioned earlier, our implementation of the ray tracer takes in a set of light sources, the original image, and an image with just the edges of the original image. For each of the light sources, we keep a 'light score' matrix the same size as the image, denoting how that light source affects each pixel in the image. This is computed by simulating light decay of each of 5000 light rays, and then combining all the light scores of different light sources in the final step.

For modeling light decay, we first calculate a best fit log curve to the intervals and then implement the curve as a function of our scores data. This allows us to display a smooth diminish in light intensity while also ensuring that the light expires at a reasonable distance from the source. Alternate approaches that we attempted were linear decay, discrete decay, and a combination of the two.

We wanted to implement our algorithm for multiple light sources because it poses another challenge for parallelization that is likely to be encountered in the real world. This additional feature required that we utilize dynamic heap memory instead of stack memory, since we would have to store information on the order of (# light sources * img_height * img_width) in memory. As an example of this requirement, our sample image of moderate size, has roughly 700,000 pixels. This means that if we have >=3 sources then our algorithm requires around 2 million bytes which is the theoretical limit of stack memory allocation for a C++ process.

Our first implementation of parallelized ray tracing parallelized across these multiple light sources using OpenMP. We mapped the scores-computation segment of the workload to parallel threads by letting a thread operate between memory location N and (N + (m*n)). This was trivial to implement, since there is no contention for memory. The problem with this first implementation is that the speedup is capped by the number of light sources present. This is particularly bad for the case of having only one light source, which is a realistic and likely occurrence.

We instead chose to parallelize across rays, which each program instance has many of, and which have decent opportunity for parallelism. For reference, we also considered parallelizing across pixels but the load imbalance per pixel as well as need for excessive communication made this approach infeasible. In this second implementation of parallelized ray tracing, we make use of the fact that our sequential algorithm iterates through rays in a for-loop. Each thread works on one ray and updates the pixels that it hits by incrementing that pixel's value by a certain amount. Unlike parallelizing across light sources, this means that multiple threads will have to modify the same value in memory. To prevent race conditions from attempting to modify a value at the same time, we also ensure that each increment is atomic. As an example of when multiple threads would modify the same pixel, imagine the pixels directly surrounding the light source. There are only 8 pixels around a light source and 5000 light rays, which means each pixel will need to be modified by at least 400 light rays. This number of rays modifying a pixel will decrease the farther a pixel is from the light source. By making the pixel modifications atomic, we ensure that our results are correct in these cases. Preventing simultaneous updates to a pixel means that there will be slow-down for pixels that have high contention.

We use a dynamic scheduling pattern for parallelizing across rays because firstly there are many more light rays than cores, and secondly the computation per light ray is unpredictable. Some light rays die off very early from hitting a barrier or the side of the image, while some span the entire image. Trial and error led to roughly 80 rays being a good task size for a thread.

We predict that the benefits of parallelizing across rays is not outweighed by the overhead from atomic updates. We also believe that this approach will work more consistently across all numbers of light rays, but may not be better than our first parallel implementation for cases that have a large number of light rays. The results will be discussed in more detail later, but we discovered that there is only an insignificant amount of slowdown from using atomic increments.

We also implement our raytracer in CUDA to make use of GPUs because parallelizing over rays with OpenMP is still yielding unreasonably slow results – up to 2 seconds for 5 light sources. According to Wikipedia, the memory bandwidth rate to and from the GPU is 15.75GB/s for PCIe 3.0, meaning it will not take a significant amount of time for our reference image of <4MB to transfer. Thus we believe that CUDA will be an effective way to speed up the computation of ray tracing. We designed kernels such that every light source corresponded to a single CUDA kernel call. Each kernel call uses m*n memory space, better than s*m*n contiguous space from the first OpenMP implementation. Each thread in the kernel propagates one light ray, and has shared access to the light scores array. As before, atomic increments are necessary to ensure correctness. No other synchronization was needed amongst threads, which greatly helps our speedup. We use device synchronize after spawning each of the light source's kernels because results are directly reduced to obtain the true color values in the next kernel call, after which a final device synchronize would be needed to port the new pixel values into the image to be output by the main program. Because CUDA has no png++ library, we had to copy image data into heap allocated data buffers.

Results:

Preface:

All speedup measurements are done in terms of the time taken for the sequential algorithm divided by the time taken for the respective parallel implementation unless otherwise noted. These time measurements were obtained by using in-code timing libraries such as CycleTimer, from the homework assignments. The baseline image that was used in all of the speedup calculations was on terra.png which is the wide 1155*649 pixel image featured on our webpage and included in our images directory. It resembled a typical animation frame which could seek to benefit from ray-tracing rendering.

Ray-tracing Results

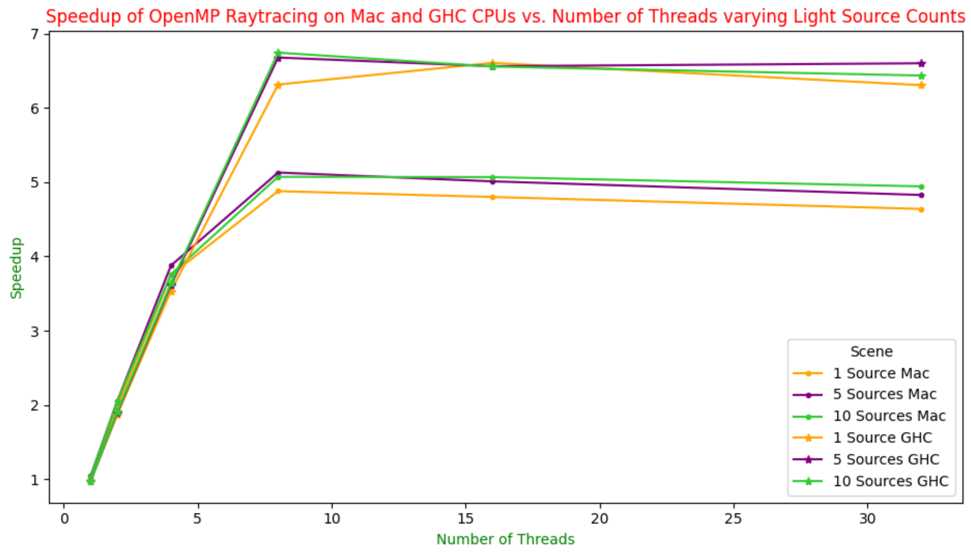


Figure 3

In Figure 3, we see a direct performance comparison between our 8-core M2 Mac CPU and the 8-core GHC Intel i7-9700 CPU machine. Both machines have similar speedups at 1, 2, and 4 thread counts, but deviate for higher thread counts. This deviation was unexpected, since the sequential and parallel algorithms being run on each machine are exactly the same, yet the Macbook air still improved in speedup at 8 cores but only to a factor of ~4.9x compared to the GHC machine's ~6.6x. Upon further investigation, we learned that Macbook M2 Air has 8 cores, 4 of which are marked as performance cores and 4 which are marked as efficiency cores. The 4 efficiency cores are less performant, but would yield better energy consumption. This factor explains the differing results across Mac and GHC.

Additionally, higher source counts provided a marginal increase in speedup. This can be explained as the call to `sumFloats` reducing the partial scores arrays into one total score array as well as the additional work of contributions from each source getting applied at each pixel. The number of CPU instructions here increases with the number of light sources, which allows multiple cores to make use of spatial locality via the default static distribution of this parallelized for loop.

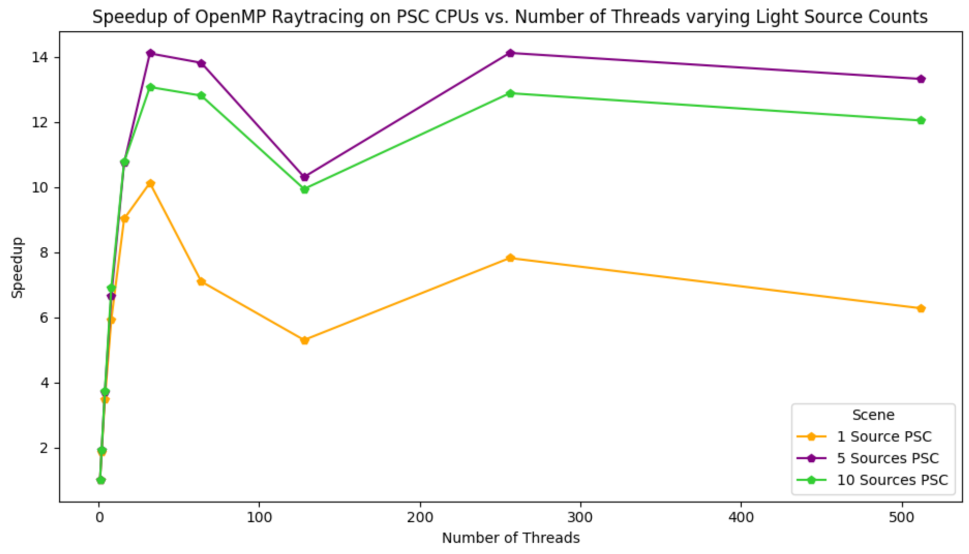


Figure 4

Figure 4's speedup plot shows that the speedup trends on the PSC machines were vastly different due to the higher number of accessible cores. We identify the maximum speedup of OpenMP as around 14x on the PSC 128-core node. Similar to how in Assignment 3 there was some slow down towards the higher core counts, the rate of atomic increments to update the partial scores matrices becomes problematic past 32 cores. The serialization of increments effectively slows down or stalls the cores so that their potential parallel operation benefits cannot be utilized.

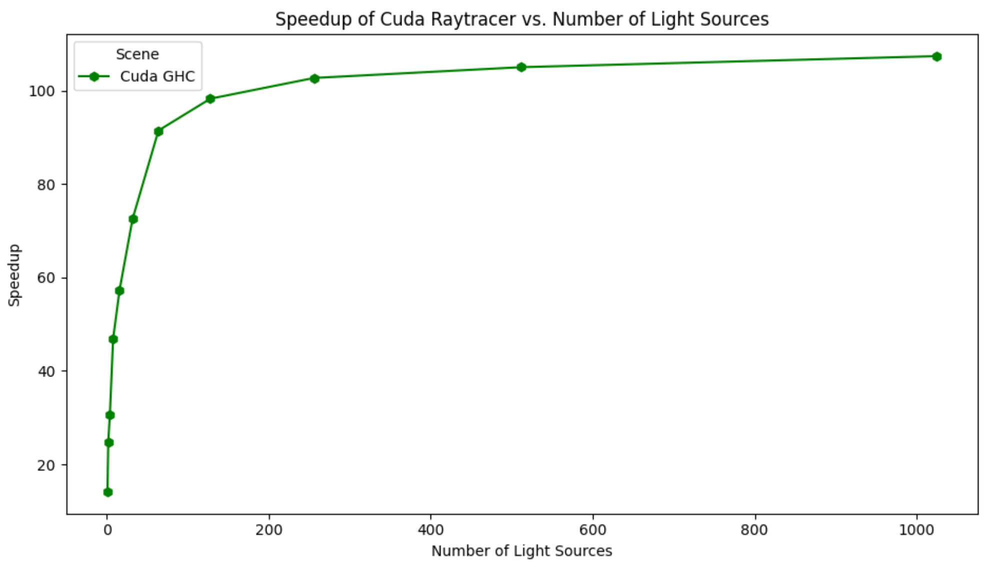


Figure 5

Since the CUDA cores have thousands of GPU workers which are not in direct control of the programmer, Figure 5 seeks to measure the effectiveness of CUDA at arbitrary light source counts beyond what would be practical. We observe that speedup begins to taper off after around 100x speedup or 128 cores. With OpenMP, 1024 randomly positioned light sources took approximately 5 minutes for the sequential algorithm to render. With CUDA, at 1024 sources the raytracer takes just under 3 seconds. Beyond this source count, the cores likely seem to run out of memory and the computed image becomes mostly black. A low number of sources took only about 2 centiseconds to run ignoring the extra data conversion operations due to CUDA not supporting png++ and C++ standard vectors, as well as the warm up time for the CUDA kernels to be accessible by the CPU. Since the convolution step was measured to take a few milliseconds after parallelizing, this meant we theoretically are within bounds to render at roughly 30-40 FPS. For reference, certain Xbox One consoles operate at 30FPS.

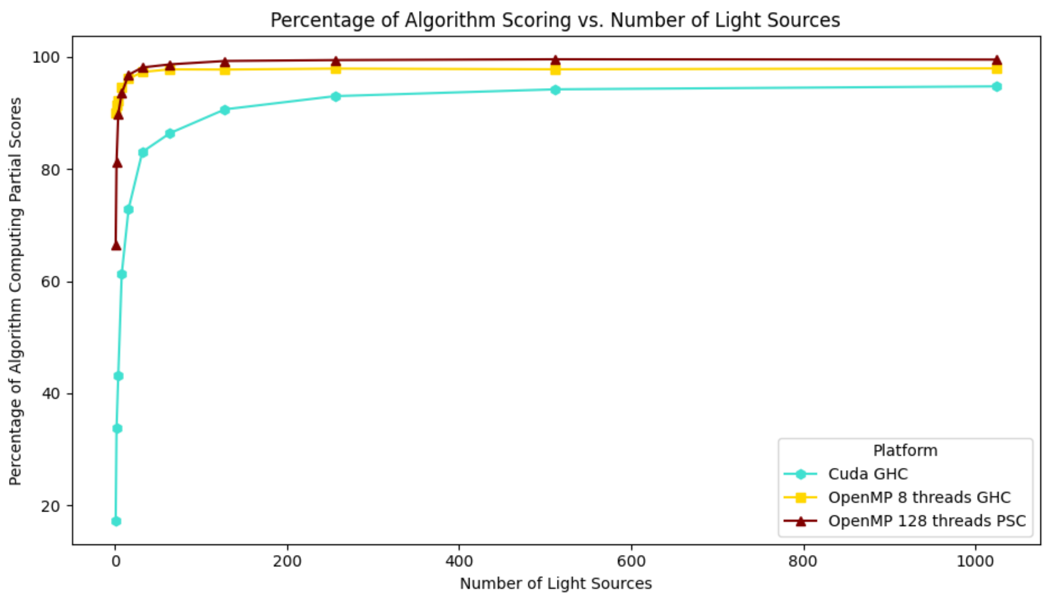


Figure 6

The ray tracer algorithm calculates a matrix of light scores for each light source and combines the score to achieve the output value for each pixel. Figure 6 shows the percentage of time spent calculating scores and percentage of time spent combining the scores. For the Cuda GHC plot, we see that the ratio of the run-time attributed to partial scores is strongly positively correlated with Figure 5's trend of parallel vs. sequential speedup, providing conclusive evidence that the propagation and scores computation portion of the algorithm is the most expensive step and that overall parallelism stems from how well that region is parallelized. We can use this to deduce that the more light sources there are, the more amount of time is spent on expensive computations. This relation is supported by the results of OpenMP GHC and OpenMP PSC in figures 3 and 4, where one light source has lower speedup than multiple light sources.

Convolution Results

Sequential Time (s)	GHC	0.0707
	Mac	0.0249
ISPC 8-wide Speedup (excluding copy time)	GHC	17.899x
	Mac	5.792x
ISPC 8-wide Speedup (including copy time)	GHC	4.202x
	Mac	2.961x

In the above table, we compare the ISPC single-core speedups with and without the time it takes to copy the png data from a png++ data structure to a 1D array. This distinction about copying time is made because if we were to make our own implementation of a png parser and png data structure, we could avoid the time it takes to copy the data. We first observe that the GHC sequential implementation time is nearly 3x the sequential time for Mac. We also observe that GHC yielded better speedup than the Mac, especially when excluding copy time.
In theory, the optimal speedup for using 8-wide vector instructions on a single core is 8x speedup, however our measurements significantly surpass this. We believe that this is the result of negative overhead in accessing the pixel data. The sequential implementation uses the png++ data structure that we mentioned stores locations of pixels instead of storing the pixels in one contiguous block of memory. For ISPC, we copy do exactly the opposite, allowing the ISPC program to make better use of spatial locality. By decreasing the time to access and store pixel information, we are able to decrease the time it takes to compute a single pixel in comparison to the sequential implementation, thereby resulting in more-than-optimal speedup.

The large speedup values also suggest good vector utilization for ISPC, which makes sense because the number of computations per pixel are exactly the same.

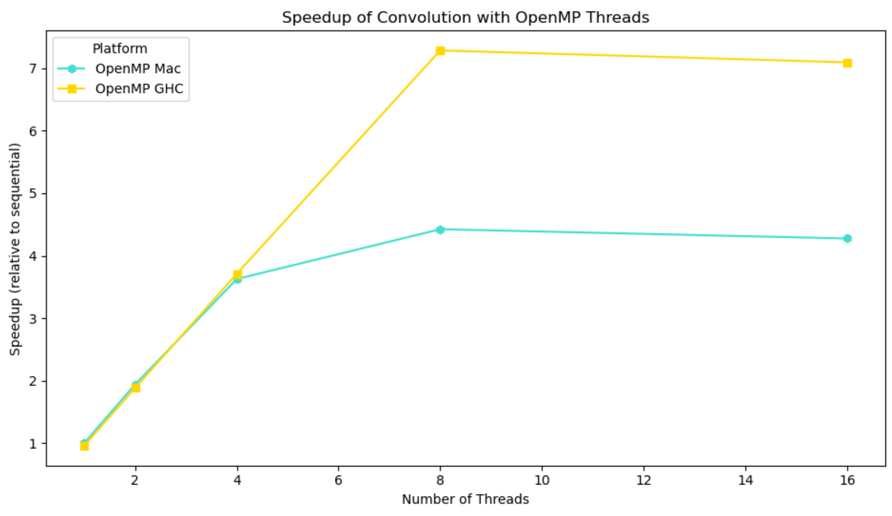


Figure 7

Figure 7 depicts the speedup of parallelizing convolution with OpenMP using various numbers of threads. Using threads to divide the work and run computations simultaneously, the ideal speedup is equal to the number of threads being used. For the GHC platform, we observe near-perfect speedup until 8 threads, and then no speedup for subsequent threads. The performance up to 8 threads makes sense because convolution computation is highly parallelizable and has almost no load imbalance across different iterations of the for-loop. We suspect that any deviation from an ideal speedup at 8 threads is due to overhead from managing the multiple threads. We see that the speedup at 8 threads is slightly less ideal than the speedup at 4 threads or 2 threads. This is in line with our expectations because for the same image, dividing the work into more threads means the time spent on computation per thread is decreased, which means that any time due to overhead will be more significant (in addition to increased overhead from more threads).

The decrease in performance after 8 threads also makes sense. The CPUs we ran our program on contain 8 cores, which means it can only run 8 threads in parallel. By trying to run our program with 16 cores, we can only achieve a maximum of 8x speedup because we are limited by the number of cores in our CPU. This, in combination with increased overhead due to thread management, explains why the speedup of 16 cores is slightly less than that of 8 cores. For the Mac platform, the results are similar to the results from the GHC platform except that there is a significant slow-down of speedup between 4 and 8 cores. As mentioned with the multi-threaded performance on Mac with the ray tracing algorithm, this is because the M1 Mac has four performance cores and four efficiency cores. The four efficiency cores will complete their work slower than the four performance cores, resulting in suboptimal performance.

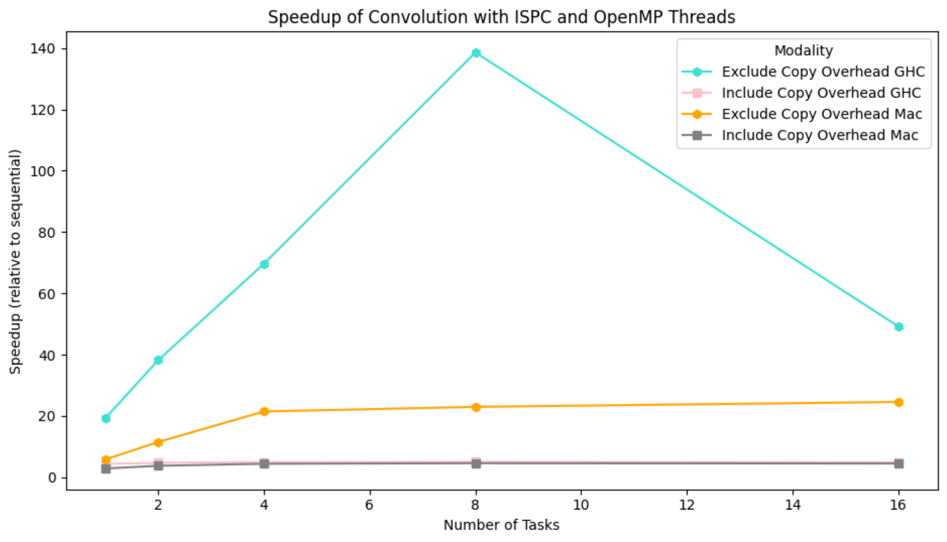


Figure 8

Figure 8 depicts the speedup of using ISPC parallelization with tasks, except the tasks are implemented via multithreading with OpenMP. This is a combination of the previous two parallelization methods, and thus sees many of the same trends. We see that for the Mac the exclude-copy speedup slows down after 4 cores because half of Mac M1 cores are more performant than the other half. In addition, we see an incredibly linear relation of the exclude-copy speedup up to 8 threads, that actually outperforms the ideal speedup of 64x with 8-wide vectors and 8 threads. We discussed this behavior earlier and identified its cause as negative overhead. Past 8 threads, there is no further performance gain from being able to utilize more cores, but there is more overhead of managing threads, so the performance decreases again.

The speedups including the copy are relatively similar for Mac and GHC at around 4x. This value doesn't increase with more cores, which tells us that the amount of time spent on copying data over is a massive bottleneck in comparison to the time of actually computing the convolution.

Sources:

https://www.wallpaperflare.com/up-movie-scene-wallpaper-154173
https://en.wikipedia.org/wiki/PCI_Express
https://www.taylorpetrick.com/blog/post/2d-ray-tracing
https://en.wikipedia.org/wiki/Kernel_(image_processing)
https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/
vec2 codebase: https://gist.github.com/acidleaf/8957147

- Timing code derived from prior homework files
- Strategy for reading light sources from text file derived from asst3 particles file parsing
- Application for live demo made use of 15-112 graphics to rapidly display images in succession

Work distribution

Emily:

Focused on the convolution side, which included finding and implementing the sequential convolution algorithm, implementing parallelism via OpenMP, implementing parallelism via ISPC, implementing parallelism via a combination of ISPC and OpenMP to simulate ISPC tasks, and collecting data about convolution speedup via ISPC and threads on Mac and GHC machines. Also contributed to the project less directly by writing most of the non-code-specific portion of reports, extensively proof reading and modifying the final report, acquiring input data for the raytracer, setting up ISPC compiling, writing code (not used in final product) for alternative image input methods such as bitmap, and attempting to use windows screen capture C++ libraries to create a live interactive demo (also not used in final product).

Rahul:

Ideated the concept of a 2D ray tracer, implemented the ray tracing algorithm, devised best light application strategies, implementing parallelism across light sources and rays via OpenMP, implemented parallelism via CUDA, and collected data on ray tracing speedups on Mac, GHC, and PSC machines. Collected further data on which sections of the program require the most time, and collected data on two different implementations of parallelism using OpenMP. Also contributed to project less directly by sorting out libraries such as png++ and omp, adding makefiles (for testing on Mac, GHC, and PSC platforms), creating a plotting script, and pulling code for a python application that can display images in real time for our demo.

Both:

Provided each other with high level input on directions for further development, collaborated with each other to figure out solutions to technical difficulties or subpar results, contributed our relevant approaches and results to reports, and fiddled extensively with libraries and platforms that didn't want to work.

Given these factors, we think the distribution of final credit should be:
61% to Rahul - 39% to Emily