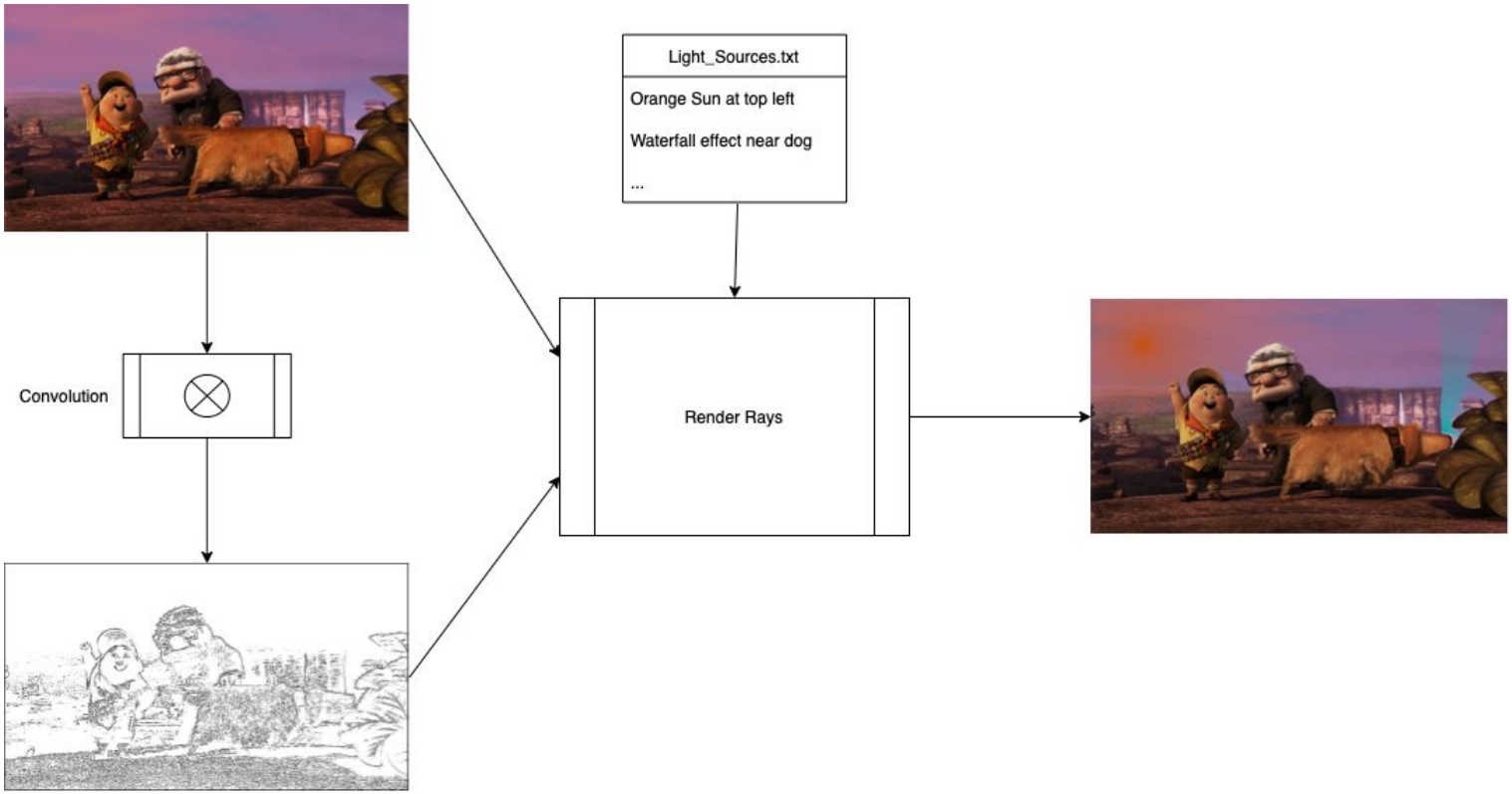
2D Ray Tracer

By Rahul Khandelwal and Emily Zhang

Idea and Motivation

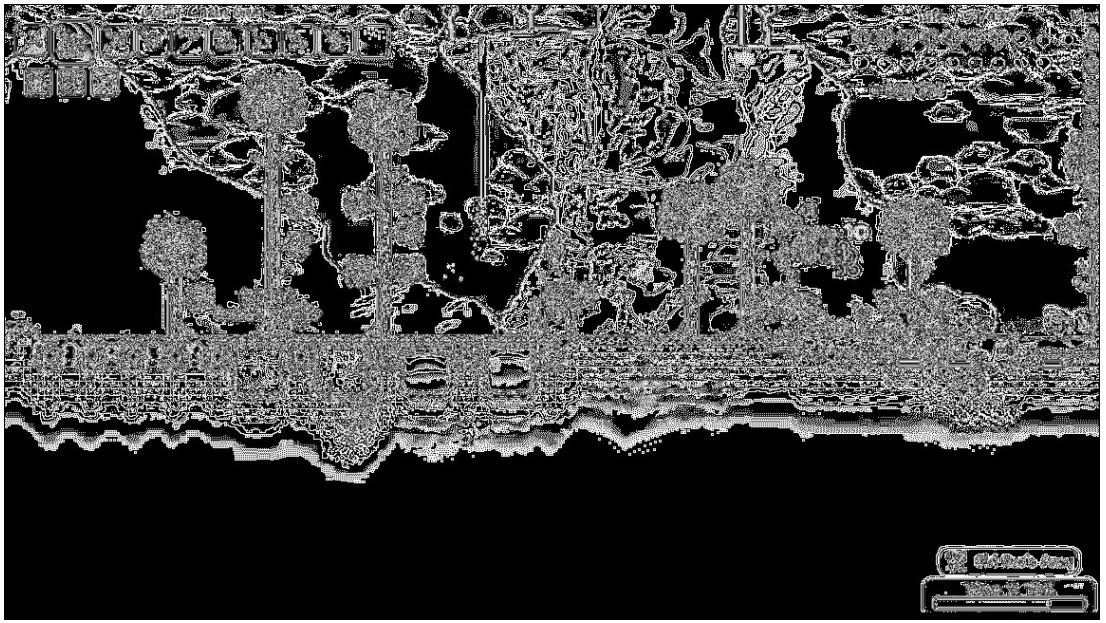
Inspired by video game visual overlays, we aim to create a 2D ray tracing program that is efficient enough to be applied to images or video in real time. We compare the results of using various parallelization methods such as OpenMP, ISPC, and CUDA for optimizing the speedup of processing the image.

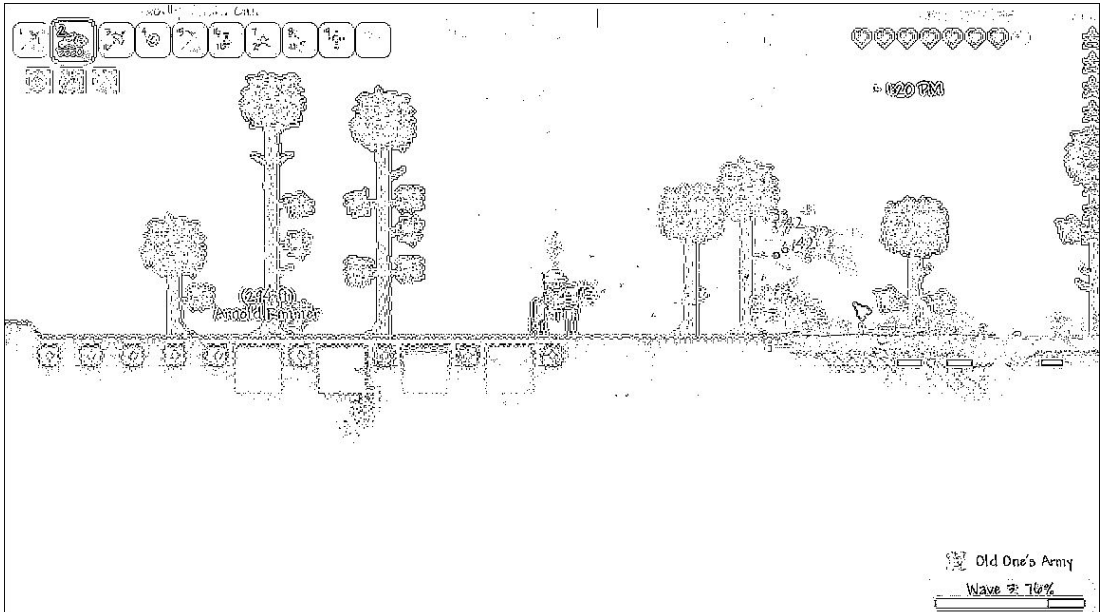


Edge Detection Implementation

- Takes in image as input, produces convolution of image
- Used an image processing kernel for edge detection (seen below).
- Iterate through each pixel and apply mask to get resulting value
- Applied adjustments such as threshold, scaling, and inversion to generate clean image for ray-tracing

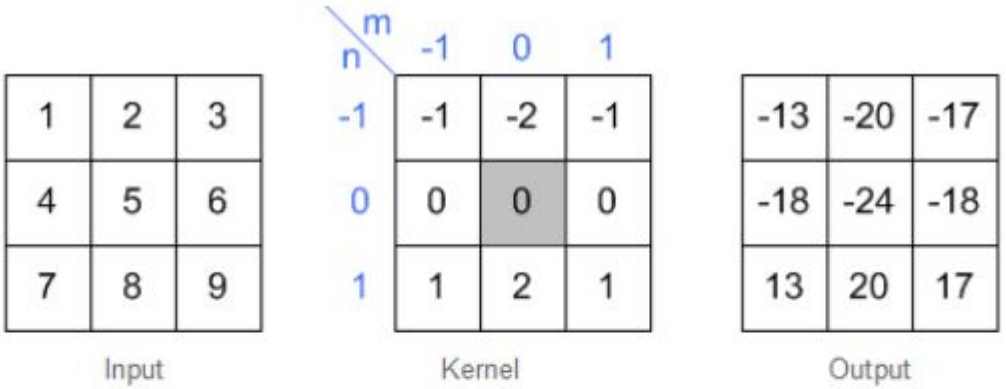
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

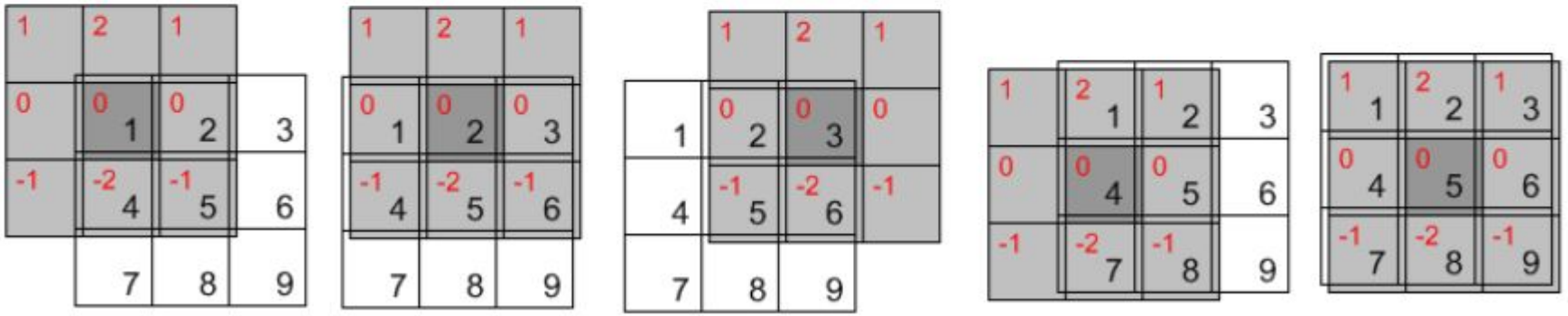




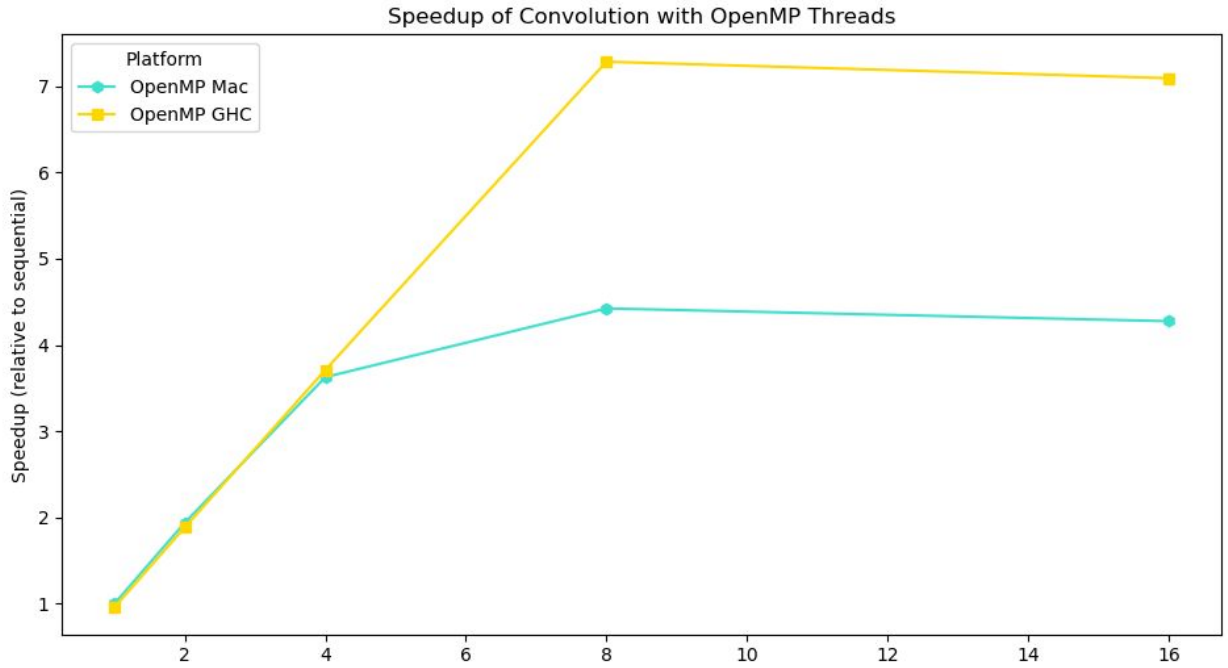
Convolution Parallelization

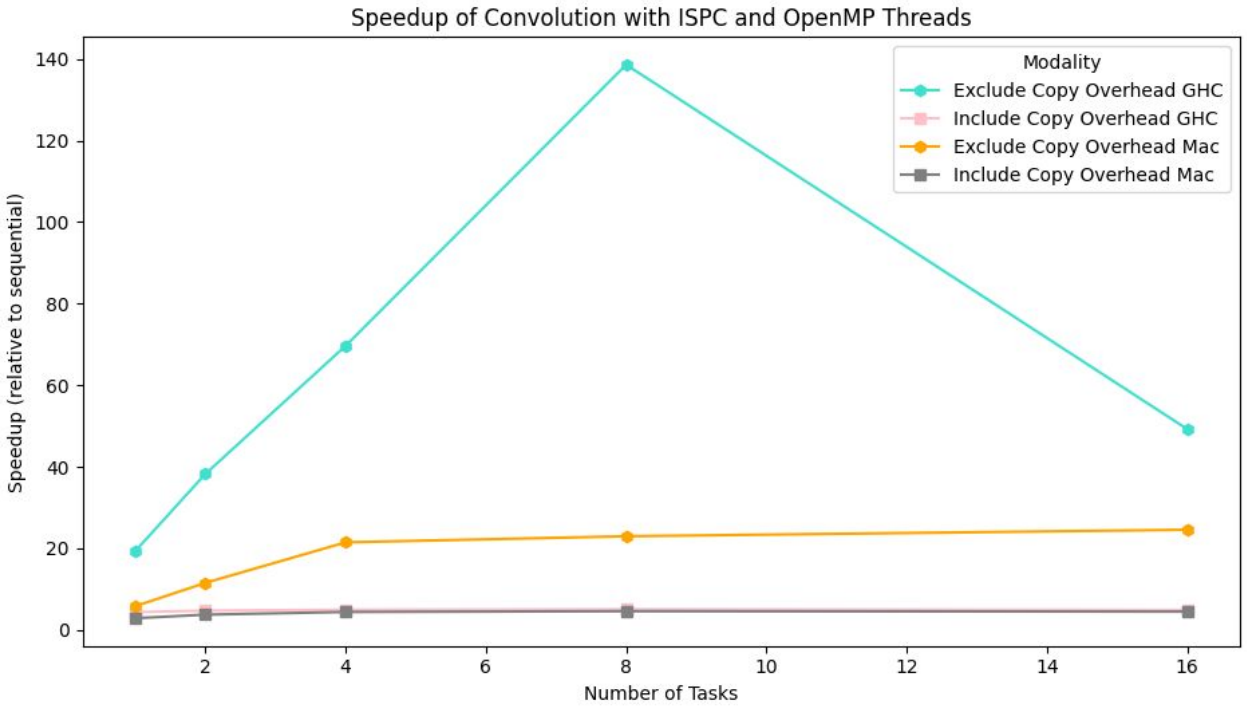
- Convolution is highly parallelizable
- Computation per pixel is not reliant on other pixels
  - Multi-threading approach promising
- Computation per pixel is predictable in length
  - SIMD approach promising





Convolution Speedup Results





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

- Observed significant speedup on a single core using SIMD vector instructions.
- Observed significant speedup using multi-core parallelism.
- Yielded best results when using ISPC on multiple cores.

Mac has suboptimal speedup at 8 cores because of different M1 core types

ISPC (exclude copy) yields more-than-ideal speedup because of negative overhead

Ray-tracing Implementation

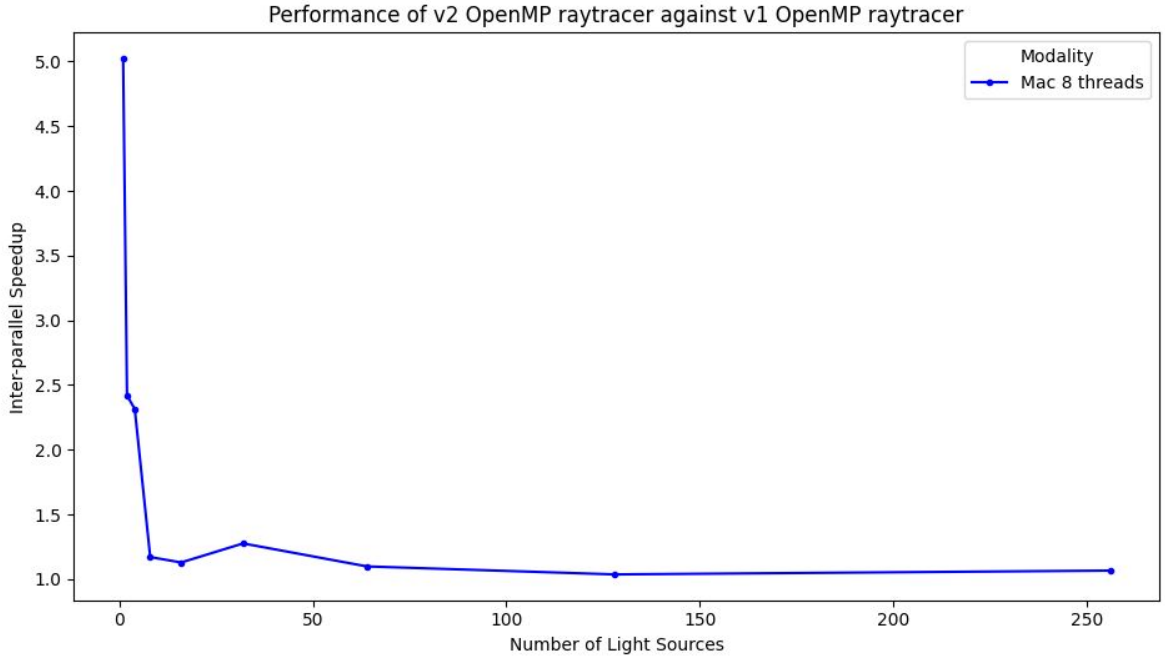
- Takes in following inputs: light sources, colored image (png), and image edges (png)
- Each light source has its own light score matrix representing its overlay onto the image
- Each light source spawns 5000 rays. A ray travels in a single direction until blocked by a black pixel in the image edges.
- Pixel light scores are updated by propagating rays. Each ray increments the light score of a pixel it visits by a diminishing amount.
- At the end, light score matrices of each are aggregated and applied to image.

Ray-tracing Parallelism

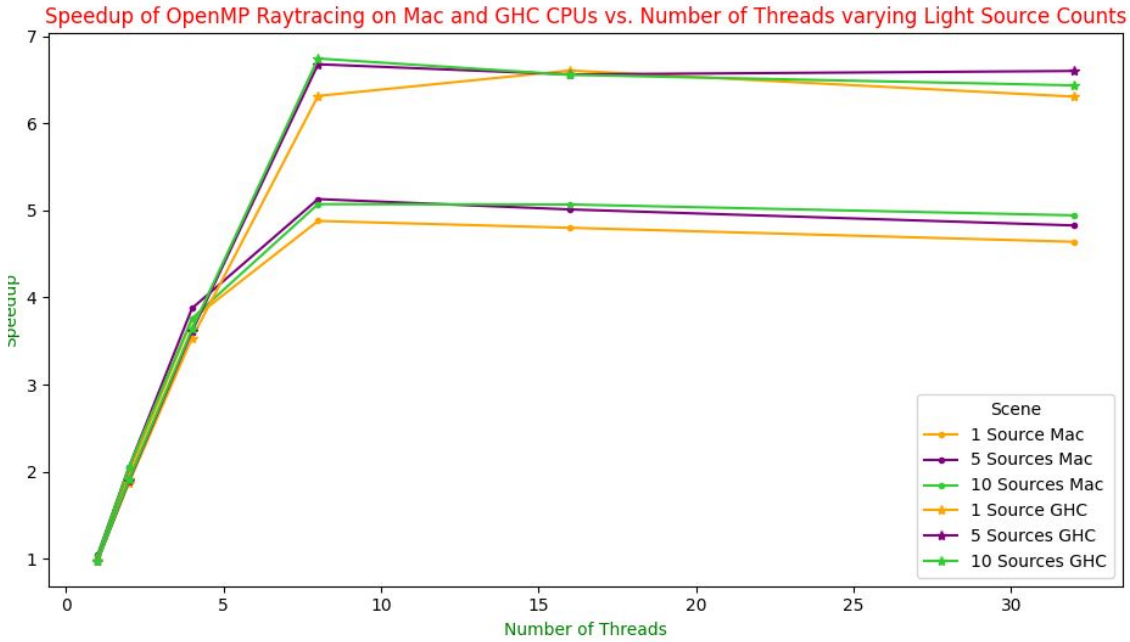
- Each light source has its own matrix of light score
- Light score matrix for each light source does not depend on other light sources.
  - Independent memory operations would benefit from parallelism
- Computing pixel value from light score matrices does not depend on other pixels
  - Can parallelize the combination step after computing light source matrices
- Multiple light rays affect the same pixel, especially around the light source.
  - Can parallelize across rays but each ray must update light score atomically to prevent race conditions. Want to choose work assignment that minimizes contention.

- Initially attempted a parallelization across sources for v1 OpenMP

- Migrating to parallelization across rays yielded better speedup compared to the parallelization across light sources in v2 OpenMP.

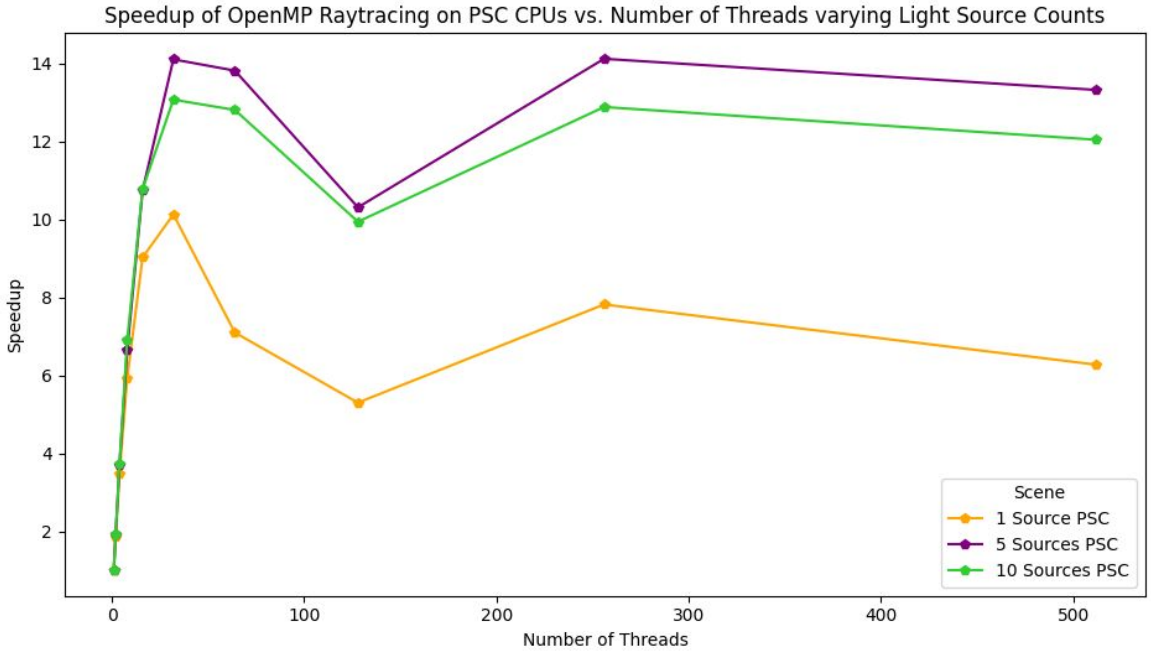


Results of Raytracing with OpenMP



Significant speedup from multi-threading
Almost perfect speedup up to 8 cores
Only 8 cores so max speedup is 8x

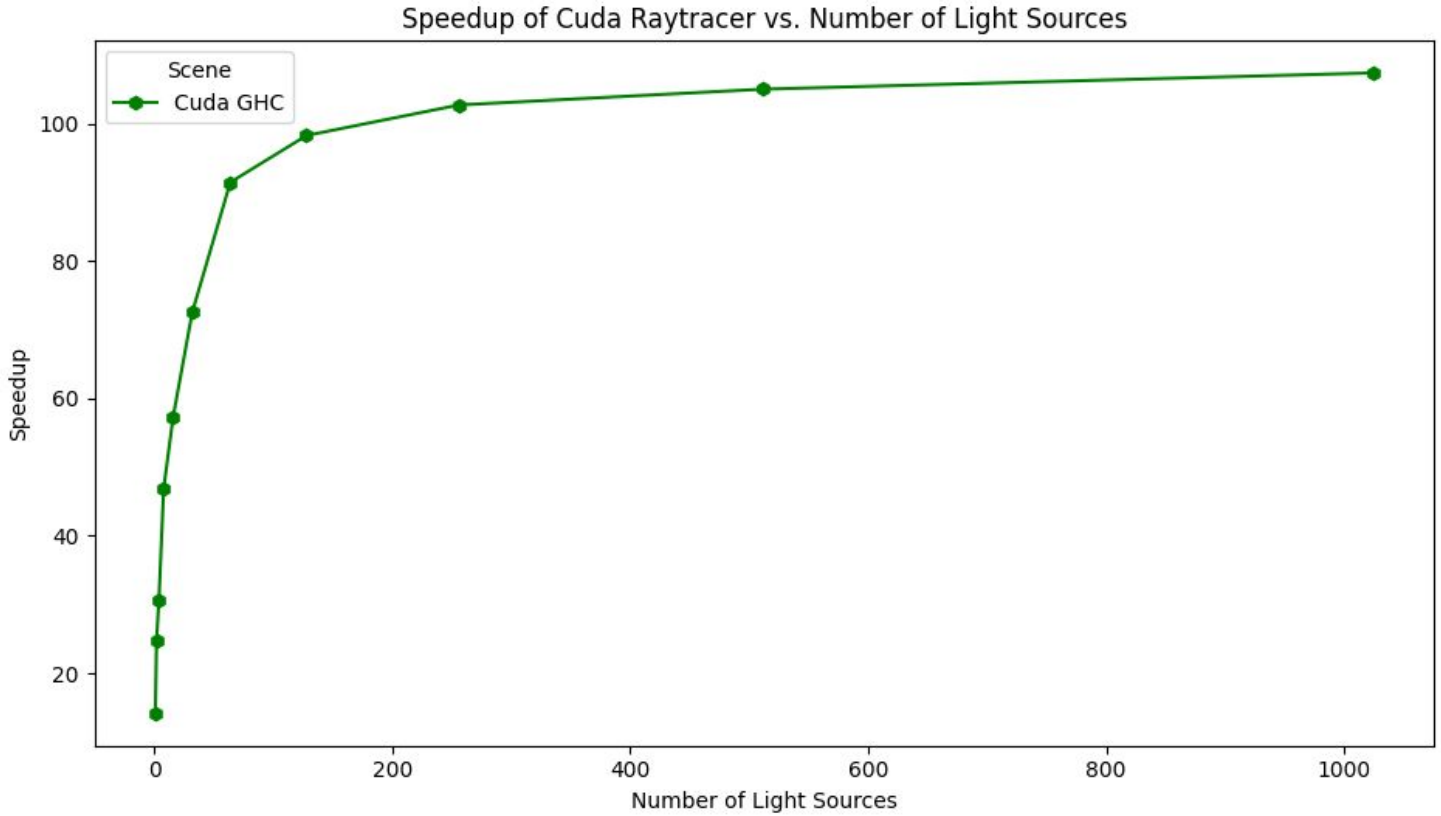
Mac has four performance cores and four efficiency cores. Performance rises less quickly from 4 to 8 cores because it uses less-capable cores.



PSC Machines able to achieve maximal speedups with CPUs, with degrading performance at super high core counts.

Number of light sources has greater impact on performance at higher core counts

Results of Raytracing with CUDA (the optimal solution)



The raytracer in CUDA had a similar underlying algorithm as OpenMP with a tweaked memory access pattern: Memory allocated in chunks rather than all at once → Higher locality per CUDA thread

Parallelization across rays enabled high opportunities for speedup, as the thousands of rays could each be mapped to the thousands of CUDA threads available in the GPU hardware.

Best CUDA time: 0.02 seconds on 1 source, under 0.05 seconds for 10 sources or less on GHC GPU.

Given that the convolution took 0.006 seconds, in practice we achieve between 17FPS at worst and 38FPS at best, with the average skewed higher as typically only 1 or 2 sources would be present.