Erick Sun
CSEN 145
10/07/2024

Parallelization of Convolution

I. Parallelization Approach

Note that this report will focus on analysis of convolution time, NOT total execution time

Approaching this code, there were many issues that must be addressed before simply applying parallel directives. The first issue that arose was the inconsistent size of images. Without a constant size, offloading batches of images onto an external device becomes exponentially tricker, as the height and width parameters become necessary for computation as well. The first task was to approach this appropriately for each parallelization technique.

For the OpenMP CPU implementation, fortunately, this was not an issue. When working on the host device, there is little reason to batch images while processing. The benefits of batching are mostly found when computational power and time must be taken to transfer data across the bottlenecked connection between devices. However, as there is no offloading taking place, there is little benefit to batching the images. As a result, the approach to parallelization is different to methods that utilize GPU offloading. The method used was to assign each running thread a section of the images within the folder. Each thread will sequentially load, then run the convolution algorithm on the loaded algorithm. With OpenMP, this can be achieved trivially by adding a *parallel for* directive under the loop that controls the preprocessing and convolution.

For the OpenMP GPU implementation[1], a different approach is used. However, since the OpenMP library is currently unable to offload any computation to the GPU, the section where convolution is performed is most likely completed on the CPU serially. However, the processing portion of this program is still written under the assumption of batching. To tackle the issue of inconsistent image sizes, the images are first loaded in, and divided into an unordered map, where the dimensions of the image is its key. This way, sending batches becomes drastically easier, as the size of the images and the length of the data array can be inferred on three variables: height, width, and batch size. Using OpenMP, the process of loading and sorting the images can be done in parallel. Furthermore, any preprocessing can be done using all cores, greatly increasing speed. After the images have been batched, the issue lies with sending the images in batch size chunks to the GPU for processing.

For the CUDA implementation, a different approach is used for batching and execution. Since CPU parallelization is not possible, it would be more inefficient to process and sort the images before offloading to the GPU in batches. As such, we approach this differently. Instead of first loading all images, we instead load images in batches, then offload the images in a given batch to the GPU. Since the image sizes are indeterminate, we must record the height and width of the images, then pass this information to the GPU as well. Once the information is sent, the GPU tackles the image convolution. The block size follows the following size (max height, max

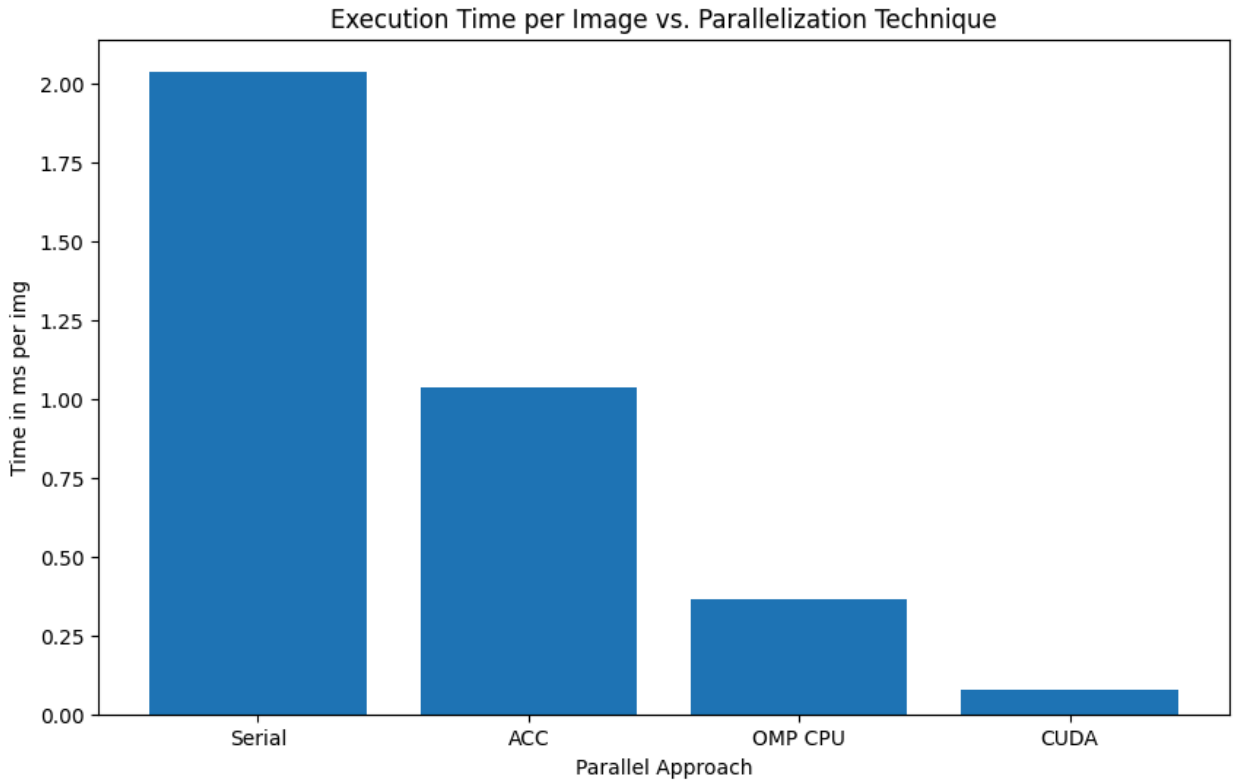
1. Note that the OMP GPU implementation is currently not function due to a failure in OMP GPU offloading

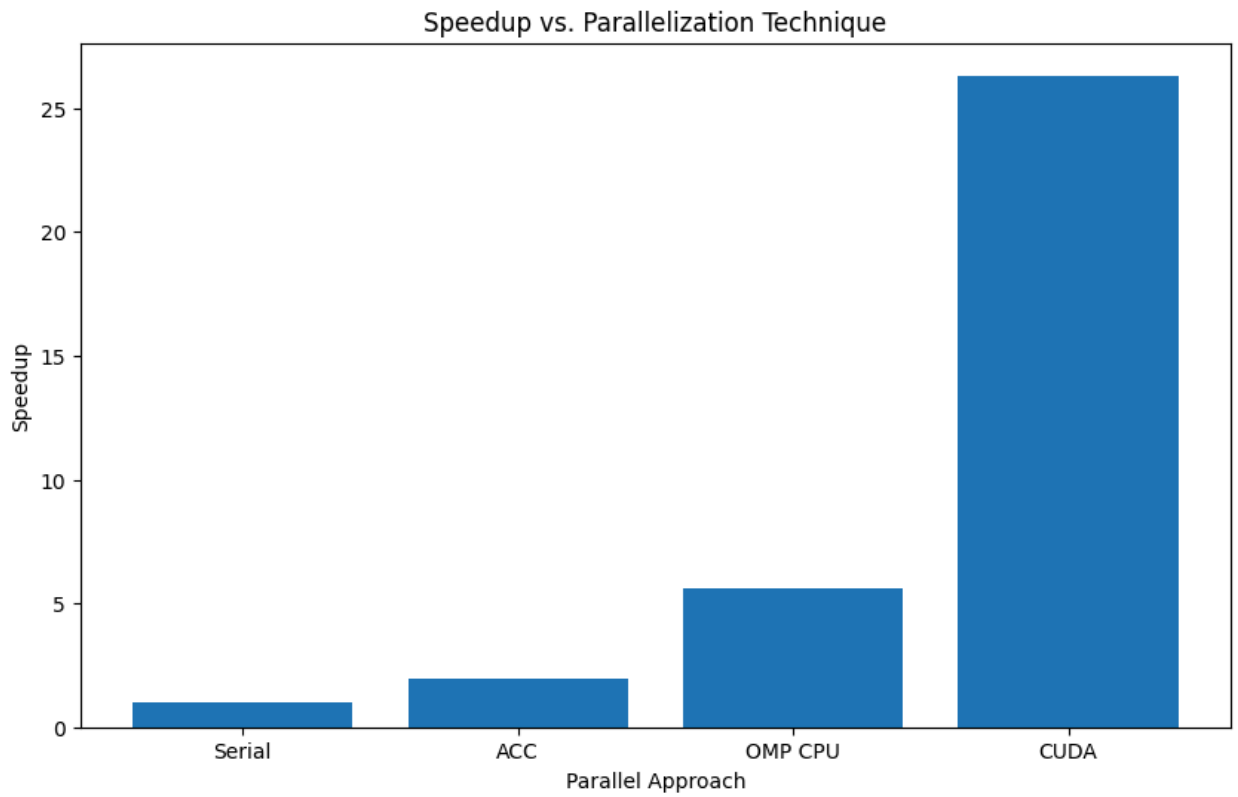
width, batch size). Then, a group of 2D (x,y) blocks is assigned to each image. From there, each thread is assigned a single pixel to perform the convolution onto. The batch size is optimized to minimize the amount of overhead created from transferring data to and from the host.

For the OpenACC implementation, I decided to approach it in a very different fashion that emphasized ease of use purely for optimizing the convolution of an image. To help facilitate faster speeds, I attempt to minimize the amount of data that must be transferred to and from the host and the device. The biggest area for eliminating unnecessary data transfer is the three dimension info arrays present in the CUDA implementation: height, width, and ptr array. To eliminate the need to transfer these, we implement an identical approach to the OpenMP GPU method. We first load the images but categorize them into an unordered map of vectors to store the data of each image. The key of each image is the dimensions of the image itself. This helps with ease of access. The next crucial step was to convert all images with a given dimension into a single one dimensional array. This allows for ease of transfer from the host to the device and visa versa. By having predetermined image dimensions, we can circumvent the need for a height and width array, and instead use a single height, width, and batch size value. This results in much faster memory copy times. The assignment of gangs and workers follows very closely to CUDA, however much of this is handled by the compiler.

II. Charts

Note that these charts represent pure convolution time, NOT total execution time





III. Analysis

As seen from the graph, all implementations achieved speedup against the serial version. However, the speedup achieved varies widely among the different implementations. The fastest implementation is CUDA. This is likely due to the fine grain control we have over the parallelization versus OpenACC and greater SIMD processing power versus OpenMP CPU. This results in great overall speedup. We are able to control the number of threads and blocks, and specify what each thread works on. This level of control allows us to ensure that no thread or memory copy is wasted. The next fastest implementation is the OpenMP CPU implementation. The convolution itself is not as computationally intensive since the images processed tend to err on the smaller side. As a result, the benefits of GPU offloading are greatly diminished. When compounded with other related overhead with GPU transfers and the lack of fine-tune control with OpenACC's implementation, it does seem feasible that the OpenMP CPU implementation could be faster than the OpenACC implementation.

Further testing from the OpenACC and CUDA implementations relieved an interesting middle ground for when it came to batch sizes. It seemed that very small batch sizes (<64) often resulted in the worst performance. This is to be expected as smaller batches result in a higher number of memory transfers from the host to the device. However, it seems that after a certain batch size, the difference in speed seems to be negligible. The performance difference between a batch size of 512 and 1024 are nearly indistinguishable in both the CUDA and OpenACC implementations. However, this difference may become more pronounced when a larger data size is introduced.

Furthermore, the GPU implementations will likely scale much better with larger sizes of images. With their high SIMD capabilities, a GPU will likely keep up much better. The GPU's strong suit is the ability to do simple calculations on a large amount of data. This is why batching is a very efficient method to approach GPU offloading. As such, as the image sizes grow, it is more likely that the CPU will not be able to keep up and the GPU implementations will prove superior. However, with this dataset, it is likely that it is not sufficient in size to differentiate the GPU and CPU implementations. Overall, GPU implementations prove to be much more efficient in tasks such as convolution. The large amount of cores in each GPU allows for large scale simple repeated calculations. However, it is necessary to balance the overhead costs of offloading to the GPU with the speedup benefits from the GPU itself.