Parallel Programming Project: Image Processing

Benjamin Barnes, Richard Northen, and Stephanie McCumsey

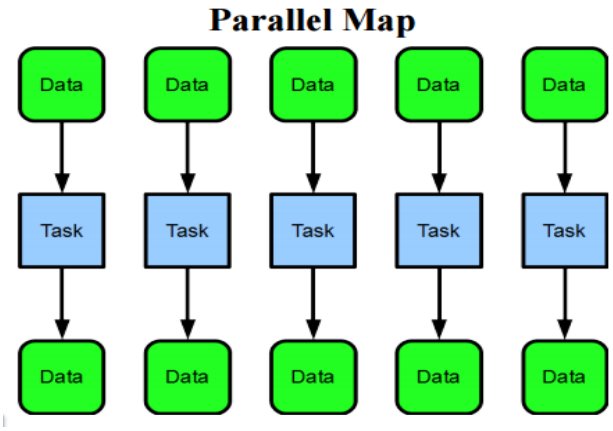
CIS 410
Spring 2015

PROBLEM DESCRIPTION

Large scale image processing is a problem that is becoming more evident in the age of 'big data' - as the data set grows, the processing time grows with it. This is particularly problematic when the available data grows exponentially, or even faster. In a particular example, training and testing a convolutional neural network can use over 9 million labeled images[1]. Pre-processing these images has been proven to noticeably decrease the errors made by the network[2] and network's accuracy can be improved when trained with larger sets of data. Thus as GPU memory expands and the datasets of labeled images continue grow, so will the need to be able to pre-process images efficiently. By taking advantage of available parallelism in the data as well as the image transformation tasks themselves, we can contribute to a significant speedup in the initial phases of a machine learning pipeline and circumvent the problems associated with processing big data sets.

PROJECT OUTLINE

By using JPEG metadata, image processing libraries, and the images themselves, raw images can be normalized to a set size and type. Additional transformations applied onto these standardized images with these tools can cause features within the image to become more discernible as well as enabling them to be compared with other images in the set. This means that an otherwise unrelated set of images could be ranked, sorted, or categorized in a relevant manner. Due to working with large sets of images, we were able to process them in parallel without fearing parallel overhead - once the serial portion of work has been completed (enumerating the set of images), the rest of the work to process them can be run in parallel. Furthermore, by using patterns of parallel programming we were able to parallelize the image processing tasks themselves. By utilizing this combination of data and task parallelism we predict a speedup in our image processing programs as compared to their serial counterparts.



METHODS

This project carried out simple metrics with OpenMP to manipulate images in parallel. Alongside the open source computer vision library OpenCV (mainly used for I/O), we applied three different patterns of parallel programming to manipulate images. These patterns take advantage of multiple threads and workers that a similar serial implementation is not designed to work with and therefore cannot use. The three patterns we use to invert pixels, overlay two images, and blur and sobel a picture are the map, map-reduce, and stencil patterns respectively. The specifications for our machines can be seen in Table 1. OpenMP allows us to code in parallel (as allowed by the hardware) and it provides a timer used for our measurements (via command `omp_get_wtime()`). The hardware provides insight into bounds observed in our performance while the parallel software allows us to take advantage of the parallelism available to us for our image processing algorithms, as

Table 1. Machine specifications (provided by TAU)

CPU Cores	4
CPU MHz	1238.132
CPU Type	Intel(R) Xeon(R) CPU E5-2603 v2 @ 1.80GHz
CPU Vendor	GenuineIntel
CWD	[redacted]
Cache Size	10240 KB
Command Line	[redacted]
Executable	[redacted]
File Type Index	1
File Type Name	TAU profiles
Hostname	cerberus.nic.uoregon.edu
Local Time	2015-06-10T16:44:18-07:00
Memory Size	16382116 kB
Node Name	cerberus.nic.uoregon.edu
OS Machine	x86_64
OS Name	Linux
OS Release	3.18.9-100.fc20.x86_64
OS Version	#1 SMP Mon Mar 9 16:27:23 UTC 2015

described in the next section.

PARALLELISM IN PRACTICE

The problem size that we will be working with is not necessarily increasing as time passes (although we could potentially make it by automatically grabbing new images to work with), so our speedup reflects Amdahl's Law (fixed-size speedup) rather than Gustafson-Barsis' Law (scaled speedup). As far as parallelism, there are a few way we could parallelize:

- *Data Parallelism*:
  Process each image individually. We would use parallel algorithms to interpret this photo's data and process each image sequentially, similar to a queue. This would allow us to update and display results as each image was processed, rather than waiting for them all to be done.
- *Task Parallelism*:
  Process all the images in parallel. Each thread would be executing a serial algorithm to process the image it was assigned. Assuming that the photos are of similar size (as far as data), this would lead to all processes finishing at about the same time.
- *Data & Task Parallelism*:
  We could potentially have multiple images being processed in parallel as well as the algorithms processing them running in parallel. Although most likely slower than data parallelism, since each there will be less threads available per image, this would allow us to see results of the processed images once it has finished.

RESULTS

MAP

The Map pattern was the first image manipulation filter we implemented. This is one of the most basic programming patterns as it is primarily used on embarrassingly parallel problems - that is, all the processing for each bit of data can be done individually. Assuming that the given data set is relatively large, this pattern is usually limited by the hardware provided. In our case, mapping the pixels in a photo can range from 500,000 operations (.5 megapixels) to 70,000,000 operations (70 megapixels) and even much larger. This is why a majority of parallel image processors utilize the GPU when manipulating or gathering information about a photo.

The Map patterns we implemented consisted of a color to black-and-white and an inverted colors filter. Both of these were relatively simple to implement algorithmically, as they just iterate over each pixel and perform some computation on the RGB values and saves the modified values.

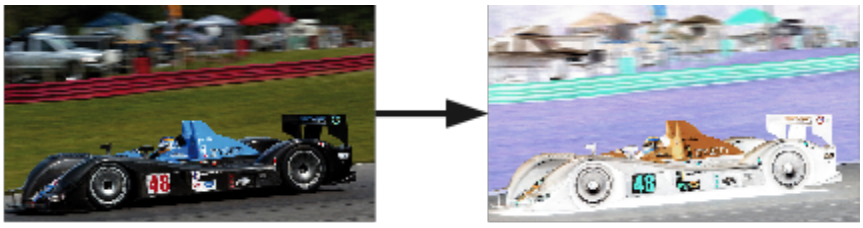


Figure 1, Example of "map" pattern

Implementing these simple filters and testing them on photos of various sizes allowed us to get an idea of how WOPR would perform with these embarrassingly parallel algorithms. Figure 2 shows the performance measurements on WOPR when using the invert color filter on a single photo of various sizes. The X-axis shows the size of the photo in megapixels, ranging from 1 to 70, and the Y-axis shows processing time in the first graph and speedup in the second graph. Using photos of various sizes allowed us to make some observations. We noticed that as the size increased, the speedup gained decreased, and eventually the performance gain was minimal. Looking at the graphs, it is clear that past approximately 10 MP, the speedup is negligible when working on WOPR.

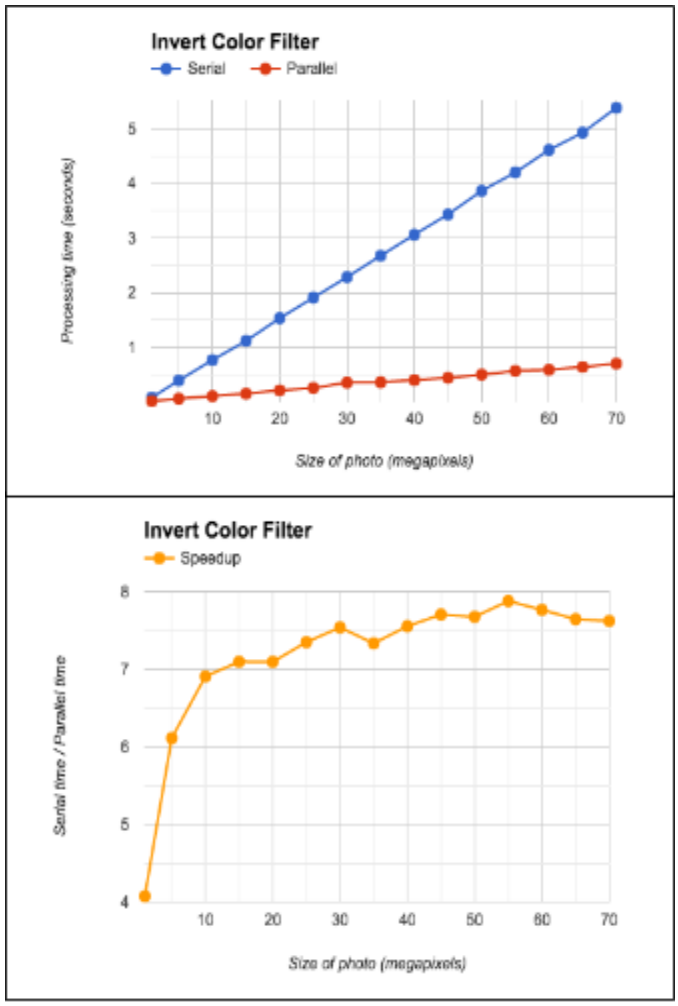


Figure 2. Performance of the map pattern

MAP REDUCE

After implementing the map, we wanted to implement a map reduce, which would allow us to work off the previously implemented pattern. Although this pattern is not used too frequently with machine learning, it does have particular use cases, such as overlaying two or more photos to see a potential match. The map reduce pattern we implemented took two photos of the same size, multiplied their RGB values by a factor of .5, and then added them together. The result of map reducing two photos with our algorithm can be seen in figure 3, where two houses have been merged together. While our implementation only merged two photos, this could easily be ported to merge any number of photos.

Another application of the map reduce pattern is iterating over each pixel and having a separate sum for each of the RGB values. Once we have iterated over the whole picture, these sums could then be divided by the total number of pixels giving us the average RGB value for each pixel.

Figure 3. Example using "map-reduce" pattern

STENCIL

The stencil pattern is used very frequently with image processing, as each pixel is usually concerned with its neighbors rather than just itself. The first filter we made that implemented the stencil pattern was a blur. This algorithm allows the user to specify how large they want the stencil to be: the larger the stencil, the greater the blur. An example of our blur algorithm can be seen in figure 4. After the blur filter was working, we then did very similar performance tests on WOPR as were done with the inverting color filter (map). The results were almost identical the the graphs in figure 2 - the main difference being the processing time. Because the blur stencil must average the values of its neighbors, there is many more computations being performed, so one would expect the overall processing time to increase. The linear processing times were apparent and the speedup of the blur filter hovered around ~7 for any photo that was 10MP or larger.

Another stencil (or kernel - a kernel being a matrix surrounding a pixel, so in this case a 3x3

Figure 4. Example of “stencil” pattern to produce a sobel (left) and a blur (right)

kernel is a 9 point stencil) algorithm we implemented was the sobel kernel. This is a frequently used filter for image processing in machine learning as it is used to detect edges in a photo, which can then be interpreted by the machine learning algorithm. The sobel filter uses two kernels to approximate derivatives, which are then compared to the original picture. Obviously this ends up being even more computations than the blur filter, but because we are working with data that has no dependencies, we can execute everything in parallel.

BATCH PROCESSING

After we had implemented the parallel image filters, we wanted to see where the parallelism is best utilized when processing batches of photos, as this is exactly what a machine learning program would have to do. The first thing we tried, to set a baseline, was to run everything sequentially. In this scenario, we are processing one image at a time and one pixel at a time. As one would expect, this ran the slowest. When processing 50 5 megapixel photos (inverting the colors and applying a blur), the serial version took 402.308 seconds. After we did this, we

processed one photo at a time but used the parallel image filters that we created. Again, we inverted the colors and then applied a blur, but this time the filters were manipulating multiple pixels at the same time. When we processed the same 50 photos as before, our runtime dropped down to 62.9466 seconds, resulting in a 6.39 time speedup. This method of processing can be viewed in figure 6 - the graph on the right. After processing the photos in this fashion, we tried the method of processing images in parallel but having the filters applied sequentially. This method of processing can be seen in figure 6 as well - the graph on the left. When we processed the same set of 50 photos, our runtime was 55.1543 seconds, a 7.29 time speedup when compared to the sequential version, giving us the fastest processing method. This method is likely the fastest because we are only starting a series of threads for the whole batch of photos, while when we are parallelizing the filters, the threads must be started and stopped for each image.

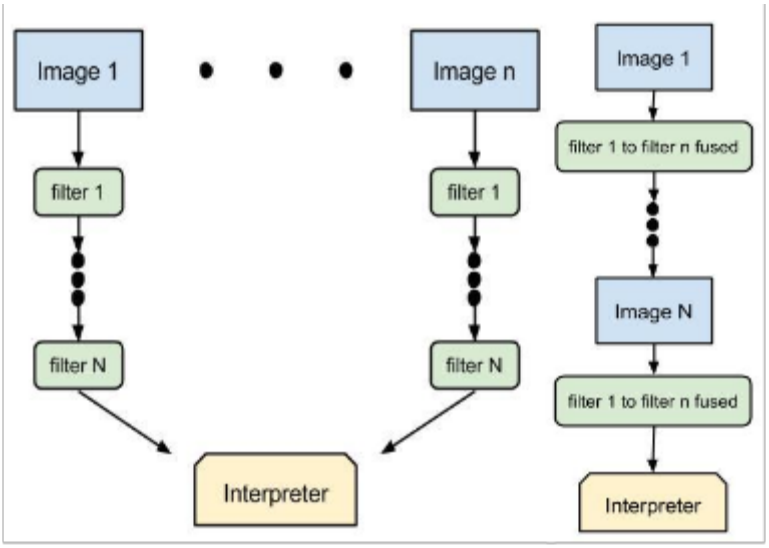


Figure 6. Processing 50 5MP photos with batch processing and pipeline results:
Serial: *402.308* seconds
Left: *55.1543* seconds
Right: *62.9466* seconds

One surprising observation when running on WOPR is that when we tried to run our programs with more than 8 threads, such as 16 or 32, the speedup was not as great as when using just 8. This is likely because of the hardware on WOPR and that introducing more threads resulted in unnecessary overhead.

Unless one is processing a few pictures that are very large, the parallelism is best utilized on a per photo basis first. One interesting method that could potentially run the fastest (assuming one has appropriate hardware) is to allows the filters to run in parallel only when the hardware has available resources. For example, if one is processing 200 photos but has 256 cores, 200 of these cores are allocated to the pictures first but 56 of these cores are unallocated. Giving these to the photo filters could potentially make the program run faster but as we saw before, it could also result in a slower program as the overhead of starting and stopping threads can be significant.

FUTURE WORK

Due to the of the nature of our problem involving few dependencies we predicted that utilizing a GPU would have sped up our programs even further. This is because the various filters that we were using were fairly light-weight and were repeated for every image. The filters could all be loaded onto the GPU and stored there as each image to be processed is read in, processed through the filters, and then written out to a disk. Also, both aspects of our code were parallel - the data set of images were processed in parallel and the filters themselves were also able to work in parallel so there would be plenty of kernels that could be generated for the threads.

Our focus for this project was on processing images. It has been shown that preparing images by pre-processing them before feeding them into a neural network decreases the amount of errors and parallelizing this step has significantly reduced the time it takes. A next possible step would be to take a look at the neural network itself. We were considering looking into machine learning and how we could extend our project with it. The training process of the network involves feeding a large set of data into it and there may be opportunities for parallelism here. If we could obtain a significant increase in performance with a parallel implementation, we could substantially increase the amount of data we can feed into the neural network. This means that within the same amount of time, a parallel implementation could process more data than a serial implementation. Not only would the resulting network be more comprehensive, but the images would have been pre-processed to allow for even more accuracy.

SUMMARY

This project shows that after taking advantage of parallel programming patterns we observe better performance. As parallel hardware grants us more memory and workers to utilize, limitations such as image size become obsolete as we're able to simply increase the number of workers. Images can thus be processed faster to produce various transformations to be fed into other pipelines, such as training for convolutional neural networks. We noticed during our project that image processing has many opportunities for parallelism. Each image can be processed individually, with a high degree of concurrency within a set of images. An image can also be broken down a variety of ways. In some filters, such as the invert color and overlap, each pixel can be individually processed. Other filters, such as the sobel and blur, use a stencil pattern and must be parallelized using other strategies.

During the course of our project, we learned the importance of parallel computing. Many applications have the potential to be parallelized - with even more potential for speedup. There is a certain amount of work to be expected beforehand. Opportunities for parallelism must first be located, if they exist at all. Appropriate patterns must be applied to each area of the program with careful consideration of any issues that can arise.

REFERENCES

1) Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada, 2012

2) Y. Le Cun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, L.D. Jackel, et al. Handwritten digit recognition with a back-propagation network. In Advances in neural information processing systems, 1990

3) Joseph Tarango 2015, Unversity of California, Riverside, accessed 12 June 2015,<http://www.cs.ucr.edu/~jtarango/cs122a_project.html>

4) Malony, A. CIS 410 Spring 2015 - Lecture slides and discussions, 2015