Casey Adams, Kevin Connell
ECEC 622
Project 5- CUDA Blur Filter
2/22/2021

CUDA: Blur Filter

Implementation of CUDA

Some tasks, such as blurring an image, require large amounts of independent operations. For this reason, tasks like this can be run in a parallelized manner and experience significant speedups. While CPU parallelization can offer dozens of parallel threads to speed up the computation, there are other devices which can offer more. Modern GPUs are capable of processing thousands of threads simultaneously which makes them optimal for highly parallelizable tasks such as this. For Nvidia GPUs, an architecture known as Compute Unified Device Architecture or CUDA is available which allows programs to take advantage of this parallelization for compute tasks. CUDA is what is used in this lab to greatly improve the processing speed of the blur function.

The outer for loop of the blur function simply iterates over the pixels in an image, reads the values of neighboring pixels, averages their intensity, and saves the result to the pixel being processed. With a single thread implementation, this outer for loop iterates over every individual pixel in the image. This process is time consuming, particularly for large blur sizes. Instead of running over each pixel individually, CUDA allows for each pixel operation to be submitted to the GPU as a separate parallelizable chunk of code. This works by first allocating space on the GPU for the input image and the output image. Then the input image is transferred over to the GPU for processing. Once that transfer is complete, the GPU is sent the cuda program to execute (blur_filter_kernel.cu). This program will operate within 1024 thread blocks at a time for the GTX 1080 used here. This allows the pixel operations to happen significantly faster than on a CPU. Once complete, the output image data can be transferred back to the CPU. In order to properly time the execution of this code while ignoring the transfer time, a cudaThreadSynchronize() function is called which waits for the CUDA task to finish.

Performance

Table I below shows the performance results of the blur operation on various image sizes with a blur radius of 1 (3x3 convolution). The right column shows the performance difference between a single threaded CPU version of the program as compared to the multi-threaded CUDA implementation. Table II shows the same image sizes but with a blur radius of 8 pixels. **Both tables report times that do not include the time to allocate memory on the GPU and transfer the image between the CPU and GPU.**

Table I Performance with BLUR_SIZE = 1

Image Size	CPU Times (ms)	GPU Times (ms)	Performance Improvement
512x512	9.779	0.178	55x
1024x1024	40.504	0.115	352x
2048x2048	131.346	0.286	459x
4096x4096	452.545	0.933	485x
8192x8192	1652.486	3.420	483x
16384x16384	7166.139	12.839	558x

Table II Performance with BLUR_SIZE = 8

Image Size	CPU Times (ms)	GPU Times (ms)	Performance Improvement
512x512	141.5	0.4	353x
1024x1024	419.7	1.4	300x
2048x2048	1767.2	4.5	392x
4096x4096	7554.2	18.9	399x
8192x8192	31558.1	79.0	399x
16384x16384	126781.8	278.6	455x

One first thing to note about compute time is that it increases with the area of the image rather than the scale of a single dimension. For example, going from an image with side length 4096 to 8192 results in four times the computation time. This is because the number of pixels in an image increases with the square of the side length which is the area of the image. It is also worth noting that for small image sizes, the improvement seen in GPU times compared to CPU times is less significant. This is likely due to some slight overheads which could not be accounted for. It should be noted however that memory allocation and copy times were excluded from the metrics which would account for significantly more time. With memory operations included, the CPU

was actually faster for images below 4096 (data not shown). Ignoring this information it is clear that the GPU version of the code is significantly faster (about 500x) when the image size is sufficiently large.

With a larger blur radius of 8 pixels for a 16x16 convolution, the improvements can be seen on smaller image sizes and the memory overhead became far less significant in separate testing.

Conclusion

By implementing CUDA on highly parallelizable tasks, significant performance improvements can be seen. In this lab, improvements of over 500x were seen on the compute hardware available in the xunil-05 server cluster when comparing CUDA performance to a single CPU thread. These improvements were less for smaller image sizes but approached their 500x asymptote around the image size of 2048x2048. It should however be noted that this does not include memory operations needed to copy the data to and from the GPU. With those operations taken into account the GPU remains slower for smaller image sizes. This of course is partially due to the blur size being small (1). With a larger blur size, the overhead of memory operations decreases.