Robert Waaser (rwaaser)
Samuel Russell (sarussel)
15-418 Assignment 2

Part 1

We got the following output from our parallelized saxpy implementation:

Overall: 23.518 ms [4.752 GB/s]

Kernel : 0.933 ms [119.814 GB/s]

Overall: 22.796 ms [4.902 GB/s]

Kernel : 0.934 ms [119.668 GB/s]

Overall: 22.185 ms [5.037 GB/s]

Kernel : 0.935 ms [119.578 GB/s]

The timer measuring the overall runtime of the program (already present) shows a much greater time than the timer we added measuring the runtime of the kernel. This is because the kernel runs in a highly parallel manner over the input arrays, where each thread only executes a few instructions on a single index. In the best case where every thread is executing at once, this results in a constant runtime O(1). The main CUDA program, however, performs a copy of two extremely large arrays from host memory to device memory, and then after the kernel completes copies the extremely large results array back into host memory. These memory copy operations are carried out sequentially and in the best case is an O(N) operation. Thus the overall time is much longer than the kernel time.

In addition, the performance of the program is constrained by the available bandwidth. Most significantly, the memory copy operations performed by the main CUDA program are moving data from main memory to the GPU memory over the PCIe bus, which is not very high-bandwidth. As shown by our bandwidth calculations the PCIe bus taps out at around 5 GB/s, and is probably a limiting factor in the performance of the array copy operations along with the sequential implementation. On the other hand, the thread operations (which involve moving data between core memory and the device memory) are operating at a bandwidth at about 120 GB/s, which is not quite as fast as the GPU's maximum device memory bandwidth of 177 GB/s. This implies that bandwidth is not a limiting factor for the kernel section's performance, although it is certainly not being underutilized.

Part 2

Achieved Performance for CUDA Implementation:

	size: 512x512			size: 1024x1024		
scene	ref	cuda	(speedup)	ref	cuda	(speedup)
rgb	1.94	0.24	(8.08x)	8.02	0.38	(21.11x)
rgby	1.05	0.22	(4.77x)	4.31	0.34	(12.68x)
pattern	4.32	0.31	(13.93x)	18.86	0.65	(29.02x)
rand10k	208.4	5.84	(35.68x)	882.7	17.0	(51.92x)
rand100k	2084.0	62.9	(33.13x)	8860.1	189.9	(46.66x)
snowsingle	255.5	5.59	(45.71x)	1006.3	13.8	(72.92x)

Decomposition:

For our solution we used fixed blocks of 256 threads each. The image is evenly divided into many square regions of fixed size, and each block is mapped to one of these regions. Therefore the number of blocks is a function of the width and height of the scene. Inside each block we decompose the problem into two phases.

Phase 1: Build a list of circles that overlap this region:

The master list of circles in device memory is evenly divided among the 256 threads in the block. Each thread checks if its circles overlap the region assigned to this block using the provided circleInBox function, and adds any overlapping circles to a list maintained in that thread's private memory. After all threads have finished checking their lists, their private overlap lists are combined into a common list by using prefix sum. The prefix sum is performed in parallel using the number of overlapping circles found by each thread and is used to determine what part of a global array to assign to each thread. All threads then proceed to simultaneously copy their private lists into a single array shared by this block.

Phase 2: Iterate through the circle lists shading pixels

Each thread is then assigned a small number of pixels in this region and iterates through the shared circle list built in phase 1 calling shadePixel for each circle. This function checks if the

pixel is actually overlapped by the circle and adds any necessary contribution to the pixel's value.

4. Our solution uses only syncthread barriers and not atomics. Synchronization occurs in between Phase 1 and 2 of each block. All threads much finish building their private circle-list before moving on to the prefix sum, and all thread must finish the prefix sum before they can start adding their private lists to the block-list. Finally all threads must finish adding to the block list before they can move onto phase 2. Each of these requirements is forced using a barrier.

5. We take advantage of circle-level parallelism in Phase 1 by having each thread check separate sections of the circle list. Then the prefix-sum allows threads to independently add these private lists into a master lists by predefining unique index ranges. Phase 2 takes advantage of pixel parallelism by assigning each pixel to only one thread. In addition, the order of updates is preserved without the need for synchronization by ensuring that the shared overlap list maintains the original ordering of the master list, and that the overlap list is applied sequentially by a single thread for each pixel.

6. Throughout the process we used the CUDA profiler and various manual tests to reason about our performance and possible improvements. Our initial approach was just to implement phase 2 to take advantage of pixel parallelism. We had 1 thread per pixel and each thread checked all the circles. This was obviously very slow since we were checking alot of unnecessary circles and duplicating work. This led us to implement the two phase model described.

We tried several different memory management strategies including dynamically allocating circle lists, or using a static 2d array, but found these to be slow and use excessive memory. We settled on the private list -> prefix sum -> master list as the fastest and most memory efficient.

Finally we tweaked some parameters (including the number of pixels per thread in Phase 2, and the order or circle/pixel work being chunked or interleaved) to obtain maximum performance. These modifications allowed us to take advantage of memory caching by accessing adjacent pixels concurrently and minimize warp divergence.