Alperen Degirmenci
CS 205 - HW 4
11/9/12

Problem 5

In $P5A$ I kept track of the 'fronts' using two masks that are the same size as the image. One of these masks is *const char* this_front* which keeps track of the front generated by the last kernel run, and the other is *char* next_front* which keeps track of the current front being generated. *next_front* gets swapped with *this_front* after every kernel invocation, and the kernel sets *next_front* (the old *this_front*) to zero the next time it's invoked.

In $P5B$, every thread takes responsibility for one item in the *const int* queue* (the *queue* stores indices of the front). The new front gets written to a mask *int* next_front* that is the same size as the image. This *next_front* then gets converted into *queue* for the next run. This is done by first doing a scan of *next_front*, and then compacting *next_front* into *queue*, hence storing the indices of the non-zero elements of *next_front*. I do not throw away the extra zeroes at the end of the *queue*, I simply pass the size of the compacted portion to the GPU and make threads with thead ID greater than this number bypass all operations. The CPU is not involved in managing the queue, so this speeds up the queue generation significantly. However, this method is not faster than P5A. The queue does reduce the workload in total, however this does not reduce the computation time per thread; in fact, computing the queue makes P5B run slower. The queue-based algorithm would be useful if we had an image so big that we weren't able to assign a thread to each pixel.

The output of both programs were compared to the output of the serial program (*Harvard_Small.png*) in order to ensure the results matched, pixel by pixel.

Performance on *Harvard_Huge.png*: $P5A = 659ms$, $P5B = 2649ms$.

Figure 1: GPU output.