GPU Boggle Solver

CS 179: Final Project
Miguel Aroca-Ouellette
06/03/2016

1. Introduction

The goal of this project was to build an efficient Boggle solver using the parallel processing capabilities of a GPU. Boggle is a word search game where the goal it is to find as many words as possible in a given grid of letters. Words can be formed by a sequential combination of adjacent letter tiles, this includes diagonal, horizontal and vertical adjacencies. Each word must be at least 3 letters long and cannot reuse the same letter tile. For this project I define "solving" a given Boggle board as finding all possible words on the board.

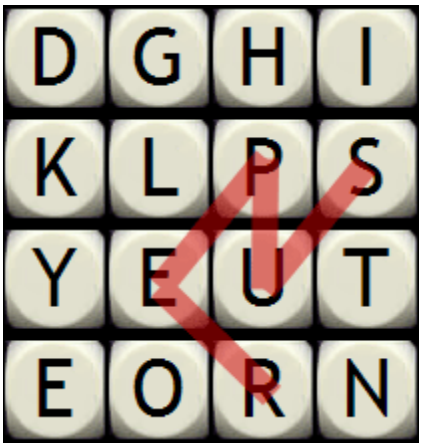

Figure 1 - Sample Boggle Grid and Formation of the Word "SUPER"

This report follows the following format: Firstly, explanations are given on how to compile and use the program, so that the reader can try the program for himself and have a better understanding of the rest of the report. Second, the structure of the Boggle solving program is explained, and the functioning of each algorithm is outlined. In total 3 different Boggle solving algorithms were implemented: the main GPU algorithm, as well as two CPU based algorithms. I then present the expected improvements from using a parallel approach versus a serial approach. Finally, the results are presented, their implications discussed, and I comment on possible avenues of improvement.

2. Installation/Usage Instructions

Note: Please read this whole section before attempting to run the program.

To use the boggle solving program on Windows perform the following steps:

1. Unzip *gpu_boggle.zip*.
2. Rename *make.txt* to *make.cmd*. (This was necessary to send the program by e-mail).
3. Open the command line in the *gpu_boggle* folder.
4. Run *make.cmd* in the command line.
5. Run *boggle_main.exe <board width> <board height>* in the command line.

<*board_width*> is the desired board width.

<*board_height*> is the desired board height.

This will by default run 10 boggle solving iterations and print the running time of each algorithm. To see the words found by each algorithm do the following:

1. In *boggle_main.cc* set *VERBOSE* to 1. (So that the CPU algorithms prints words found)
2. In *boggle_gpu.cu* set *VERBOSE* to 1. (So that the GPU algorithm prints words found)
3. Reduce the number of trials in *boggle_main.cc*. The variable is called *num_trials* and can be found on line 62. For *VERBOSE* tests I suggest setting this to 1.
4. Run *make.cmd* again.
5. Run *boggle_main.exe* <*board width*> <*board height*>.

Note that the program will print the randomly generated board, each word found for each algorithm, the total number of words found for each algorithm and the running time. It will also print if any errors were encountered by the GPU algorithm. Due to limitations in GPU memory it is recommended that the board width and height are kept relatively small, no more than a side length of 10; this limit will of course vary depending on the host system. More explanation on this is provided in later sections.

If you desire to change the number of threads or blocks per threads, the variables can be found in *boggle_main.cc* and are called *MAX_BLOCKS* and *THREADS_PER_BLOCK* respectively. Once again, due to memory limitations on the GPU, if the threads per block are increased above 10 then the GPU solver may fail for larger board sizes; the exact number will vary depending on the host machine and further explanation is provided in later sections. On the author's machine a 12x12 Boggle grid could be run using 5 threads per block.

3. Boggle Solver Program

The Boggle solving program is composed of the following 4 modules:

1. Main (*boggle_main.cc*): This module is the entry point for the program. It contains the testing and timing code as well as the main parameters. It also sets up the GPU memory, calls the GPU algorithm, and contains both CPU algorithms.

2. Boggle Environment (*boggle_env.cu*): This module contains the Boggle Board class and the Tile class. The Board class houses the various parameters defining the boggle board and defines the functions for interfacing with the board. The Tile Class is a class used by the Boggle board to keep track of letter tiles, their location, their neighbours, and whether or not they are occupied by the current word path.

3. Boggle GPU (*boggle_gpu.cu*): This module contains the GPU kernel, which implements the GPU algorithm. It also contains the wrapping function which initializes the kernel with the proper block and thread counts.

4. Trie (*trie.cc*): This module contains the Trie Class and the Node Class. The Trie Class is a tree structure which houses the dictionary trie (also known as a prefix tree). It uses the Node Class to

define the current tree structure, as well as allowing for tree building given a list of words, and tree searching given a single word.

4. Algorithms

Traditionally a Boggle board is composed of a 4x4 letter grid upon which the word search is performed. Since any properly implemented algorithm should be able to find all words within the grid given the same dictionary of words, the focus of this project was to perform this word search as quickly as possible. To increase the breadth of the problem and highlight the characteristics of each algorithm it was also allowed that the Boggle board could take on any rectangular size NxM.

4.1 CPU Algorithm 1: Naïve Word Search

This first algorithm was implemented as a proof-of-concept of the intended GPU algorithm. It also provided a naïve baseline upon which to compare performance gains from parallel computation. The algorithm is based on a naïve recursive search through every path in the letter grid for a given word. The algorithm simply involves taking a given word, searching for its starting letter on the grid, then if there is a match iteratively checking adjacent letters until the word is found (or not found). This is done in a recursive process such that at each at each recursive step the adjacent letters are checked to see if they match the current character in the word; recursion stops when the word is found or when no more letters can be checked without doubling back onto a letter which has already been used. Clearly this process is inefficient in the recursion process as it does repeated searches for starting substrings of multiple words; i.e. *care* and *careful* have matching starting substrings, but the whole word search is repeated for both.

4.2 CPU Algorithm 2: Prefix Word Search

The second CPU algorithm was implemented in order to provide a better comparison baseline for any performance gains made by the GPU algorithm. To the best of the author's knowledge, this algorithm represents the fastest approach to serial searching for words within a letter grid.

To remove the waste of repeating computation for identical starting substrings, we can preprocess the dictionary as a trie. A trie, also known as a prefix tree, is a tree which stores a single character per node and where each node is the child of a sequence of letters which come before it in a word. Figure 2 shows a simple trie formed from 11 words. The white notes are standard traversal nodes and black nodes are "word end nodes" which means that a complete word is formed when these nodes are reached. Storing our dictionary in this manner has several advantages. Firstly, it is more space efficient, as repeated prefixes are only stored a single time. Secondly, the approach is also much better suited to the game of Boggle, both intuitively and in reducing the search space.

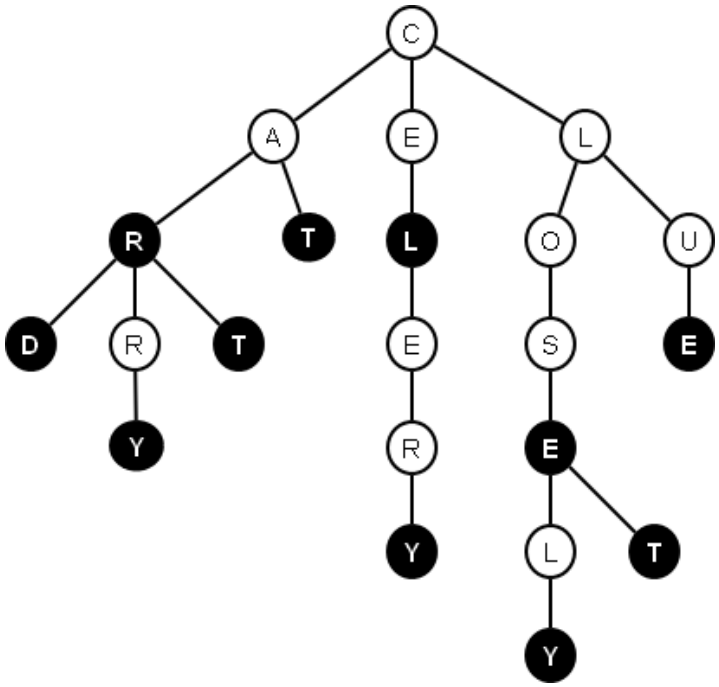

Figure 2 - Sample Trie. Black Nodes Represent "Word End Nodes"

The word search algorithm functions similarly as before, except that now each root letter on the board is assigned a specific prefix tree (which contain all the words which can be formed starting with that letter) and uses that prefix tree to spawn recursive processes on the adjacent letters which are present in the immediate child nodes of this tree. These adjacent letters are then assigned a portion of their parent's prefix tree and so on and so forth. Each recursive path terminates when it reaches a leaf node and outputs a result when it reaches a "word end node". This approach saves us the wasted computation of repeating starting substring searches. It also allows us to iterate only through the prefix tree generated for each starting letter tile, instead of across the whole dictionary.

4.3 GPU Algorithm

The implemented GPU algorithm leverages the fact that word searching can be parallelized across each word in the dictionary. For a single word this simply requires checking the $N * N$ grid for any letters which match its starting letter, then traversing the adjacent letters until the whole word is found, or the word cannot be found.

More explicitly, given some word, the board is checked for the starting character in that word. If a letter tile matching the starting character is found, then this tile is added to the current word path and the word character index is incremented. This new tile then has its adjacent tiles checked for letter matches to the second character in the word. Every time a tile is checked it is marked as such, so that the failed worth path from the starting tile to that letter tile is not checked again in the future. Obviously, letter tiles in the current worth path are also not checked for matches to the current word character. If a character match is found, then the above process is repeated on the new matching tile. If a character match is not found, then we backtrack to the previous tile-character match and continue searching. The process ends when either the word is found or we have backtracked to the starting letter tile and all of its adjacent tiles have already been checked, thus we have failed to find the word. Since the author's GPU did not support

recursion (also known as dynamic parallelism on GPUs), the kernel algorithm had to be written in a loop which maintained the state of the board and the current word path at every search step.

The above process was run in parallel with a different word for every thread in the hopes that by parallelizing across the dictionary, there would be significant speed ups when compare to the CPU algorithm. Depending on the number of threads run, and the size of the dictionary used, it may be possible for each thread to process only one or two words. As such it was expected that the algorithm would be suitably parallelizable and perform well.

5. Expected Results

It was expected that there would be significant speedups in the GPU algorithm with respect to both CPU algorithms. In fact, since the GPU algorithm is unaffected by the size of the board (except for a slightly reduced search space along the edges), it was expected that the running time of the GPU algorithm should be fairly constant for any board size. In comparison it was expected that the running time of the naïve CPU algorithm would increase at least exponentially with the size of the board, and the prefix CPU algorithm would increase approximately linearly with the number of tiles on the board; this later scenario is because the number of prefix searches simply depends on the number of possible starting tiles.

6. Results

The algorithms were run on a GTX 860M with a 1024 blocks and 5 threads per block. The reason for the limited number of threads per block is discussed in the next section. The dictionary used contained 109583 English words, of which the longest contained 32 characters. For the results shown below, the Boggle grid was kept square while the side length was varied. This is because a square grid offers the highest complexity in search paths and thus represents the hardest scenario for the algorithms; this fact can easily be seen by considering the number of possible search paths in a square 5x5 grid versus a 1x25 grid. The result of the word searches were also compared to ensure that both the CPU and GPU algorithms found the same words on the board; in all cases this was true.

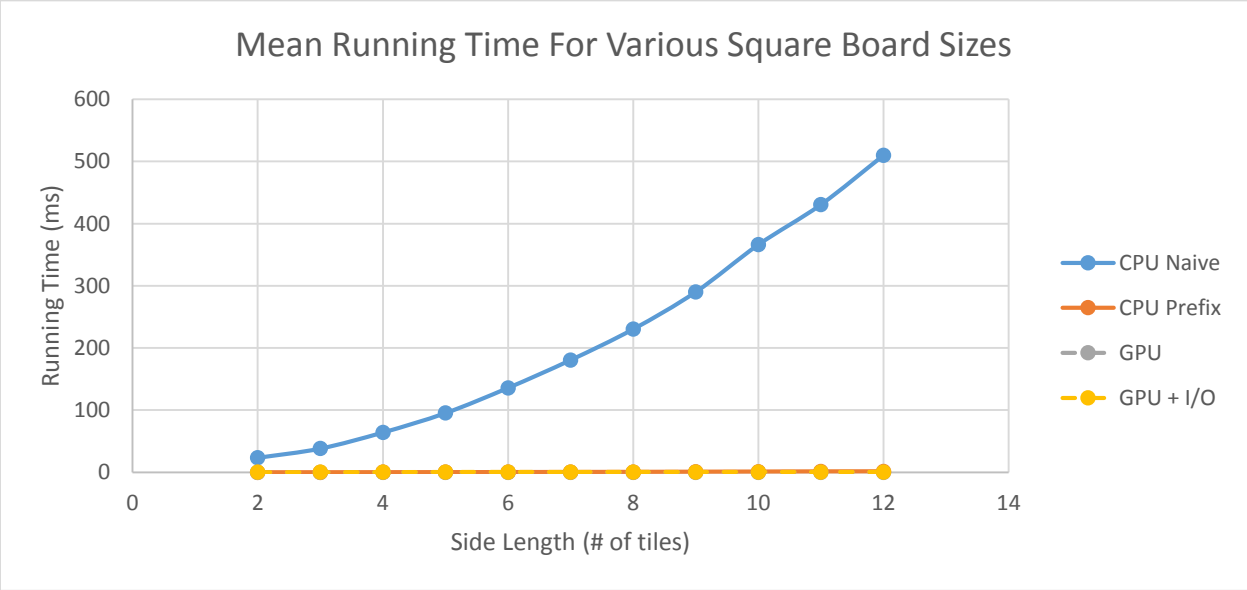

Figure 3 - Mean Running Time for Each Algorithm Over 20 Trials

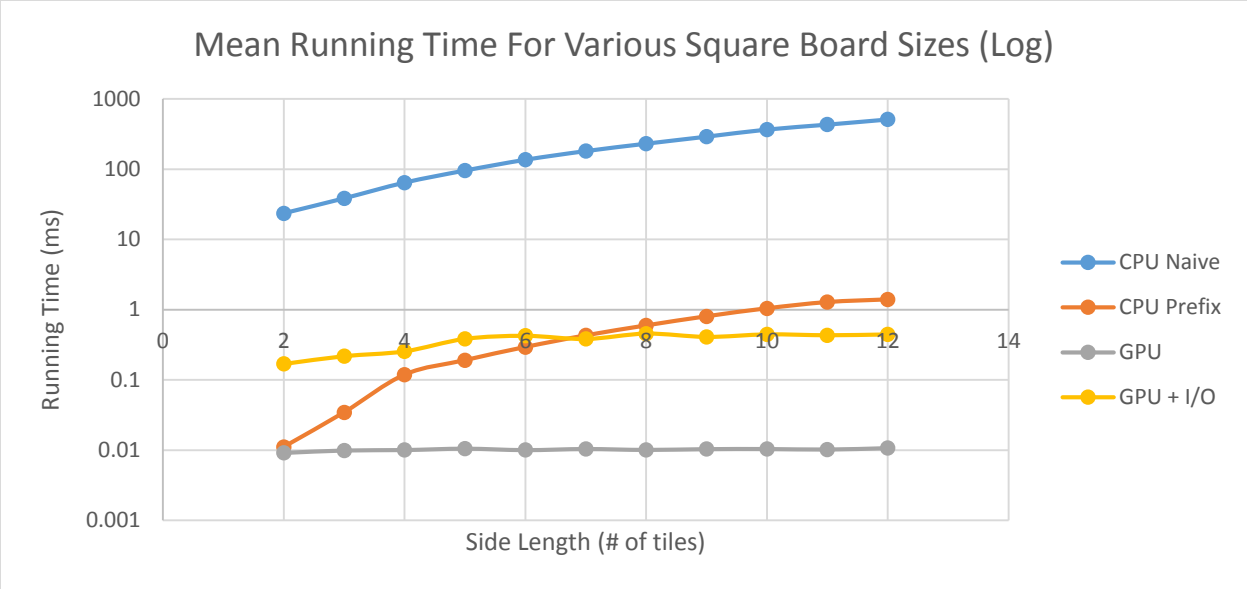

Figure 4 – Mean Running Time for Each Algorithm Over 20 Trials (Log)

Figures 3 and 4 shows the average running time of 20 iterations of our algorithm on various square Boggle boards. Each board was randomly populated with letters before being searched by each algorithm. The prefix tree building time and the time it took to transfer the dictionary to the GPU were both ignored in the figures above as the focus was on algorithm running time and not on one-time overhead. As expected the Prefix CPU algorithm performs significantly better than the Naïve CPU algorithm, and the GPU algorithm without I/O performs better than both. In addition, as was expected, the GPU algorithm without I/O has a near constant running time. The GPU algorithm with I/O, which includes transferring the Boggle board data to the GPU, has a similar trend to the non-I/O GPU algorithm, but with a constant time increase of ~0.4 seconds. As expected the bulk of the time used in our GPU algorithm is actually in data transfer. For small board sizes, this actually makes the GPU algorithm including I/O worse than the CPU Prefix algorithm; however, since the data transfer time is nearly constant it beats the CPU algorithms once the board's side length reaches 5 tiles.

It should be noted that the first call to the GPU algorithm was always particularly slow with a running time of ~3.7 seconds. However, any subsequent call would run in ~0.01 (no I/O) or ~0.4 (with I/O) seconds as shown in the graphs above. The slow running time of the first call appears to be a peculiarity of the system and may be due the author's laptop switching from the on-board GPU to the external CUDA compatible GPU (the author's laptop has 2 GPU's). As such this value was considered an outlier and ignored in the graphs above since the focus was on the mean running time of the algorithm. In addition, this outlier actually contributes relatively little to the mean when performing many testing iterations.

	Time (ms)
Prefix Tree Construction	94.7928
GPU Dictionary I/O	2116.624

Table 1 - Algorithm Overhead (Mean of 20 Trials)

Table 1 shows the overhead of the CPU Prefix and GPU Algorithms. Constructing the prefix tree is actually quite efficient, however transferring the dictionary into the GPU's memory is particularly time consuming.

Fortunately, each of these operations only has to be performed once, as they are constant given a dictionary; thus once constructed/loaded any number of Boggle boards can be solved without having to repeat this one-time overhead.

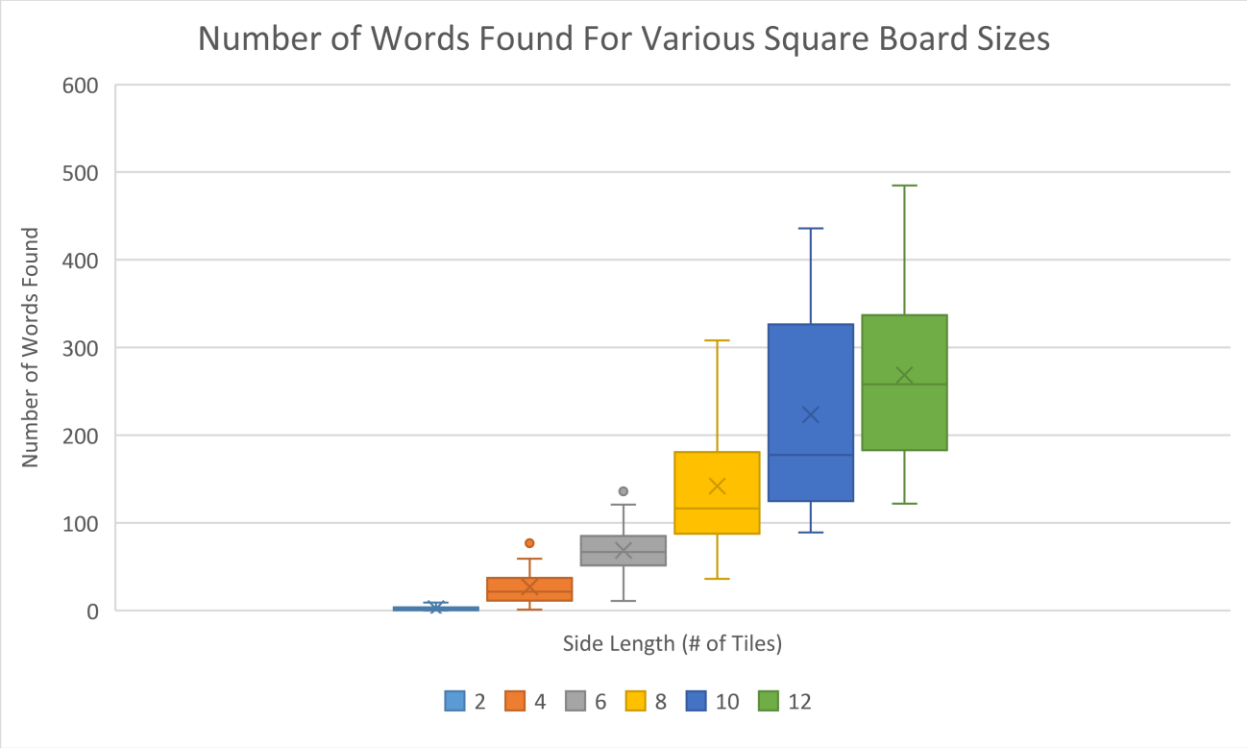

Figure 5 - Number of Words Found for Various Square Board Sizes (Mean of 20 Trials/Size)

Finally, Figure 5 shows the number of words found for square boards with different square board sizes. Clearly as the board size increases, so does the average number of words found. However, since the number of words found depends highly on the random letters generated, there is high variance for all board sizes. The above graph is identical for all 3 algorithms since they find the same words for any grid.

7. Limitations

Although the above results show the success and promise of using a GPU based algorithm as a Boggle solver, it is important to note the limitations of the implemented approach.

Primarily, the size of the board which can be solved is heavily limited due to GPU memory constraints. This is because every thread needs to maintain information regarding the word paths it has already visited so that it does not repeat the same search twice or get stuck in an infinite loop. Each thread also needs to maintain a list of its current word path and a copy of the word it is searching for. This puts heavy demands on the memory available to each thread, and since the required information to be stored scales with the size of the board (more tiles to be marked as previously travelled) the GPU can run out of memory for larger boards. As this memory is per block, this issue can be partially alleviated by decreasing the number of threads per block; hence why we used only 5 threads per block in the tests. On the author's laptop it was possible to solve a 12x12 board using 1024 block and 5 threads per block; however, a 13x13 board failed. This effect is exacerbated by the fact that the GPU also needs to maintain a massive word dictionary

in its memory as well. Both these memory issues also introduce long data transfer times, which, as seen in our results, are by far the slowest part of the GPU approach.

In addition, during implementation it was found that the problem may not have been as parallelizable as it first appeared. Firstly, it was realized that each thread is performing different operations at any given time (as each thread searches for a different word across different tile paths), and thus one would expect severe thread divergence. Furthermore, the algorithm implementation is unintuitive due to the lack of available recursion on GPUs. This can easily be seen by comparing the GPU and CPU algorithm code which, although implemented using similar search methodologies, are highly different due non-recursion on the GPU. Difficulty was also encountered in sharing classes between the CPU and GPU, as well as performing dynamic memory allocation within the kernel.

These issues suggest avenues of further work, particularly in finding ways of reducing the memory footprint of the path traversal algorithm and the dictionary. One option which may reduce the dictionary size would be to use the prefix tree approach, which should in theory be more compact. However, this approach is also more naturally implemented when using recursion, so if possible one want a recursion compatible GPU before attempting such an approach. Ideally, one would also want to find an alternate algorithm which does not require maintenance of the paths travelled within each thread, and instead simply shares the grid information across all threads; this would be the best way to reduce the memory issues.

8. Conclusion

In conclusion, a GPU based Boggle solving algorithm was successfully implemented. Although it was not quite as successful as was hoped, it still achieved significant improvements over the Naïve CPU algorithm and was marginally better than the Prefix CPU algorithm for larger boards. The memory issues which prevented its application to larger board sizes motivates modifications to the current approach in order to reduce the memory footprint per thread, as well as the high data transfer overhead.