CUDA Wordle: A GPU-Optimized Wordle Solver

Aniruddh Sriram
UT Austin
Dept. of Computer Science
aniruddh.sriram@utexas.edu

Shravan Ravi
UT Austin
Dept. of Computer Science
shravan.ravi@utexas.edu

Abstract

Wordle is a popular game where players have six attempts to correctly guess a five-letter secret word. In this project, we develop a high-performance Wordle solver[1] using NVIDIA GPUs and parallel processing techniques. Our solution takes an information-theoretic approach to evaluating guesses, where we compute the entropy for each word in a preset dictionary. We build CUDA kernels to concurrently process data and compute the expected information for each guess. With over 11k words to search, this offers a significant (12x) speedup over a serial implementation of the algorithm. We introduce two memory optimizations that accelerate runtime even more. Finally, we evaluate performance using various benchmarks and experiments. The results demonstrate the effectiveness of CUDA in solving complex computational problems with high parallelism requirements.

1 Introduction

Wordle is a word guessing game where the player has six chances to guess a five-letter word chosen at random by the computer. After each guess, the computer provides feedback on which letters in the guess match the target word - green indicates correct letter and correct position, yellow indicates correct letter but wrong position, and gray indicates a wrong letter. The objective is to guess the target word as quickly as possible.

The goal of our Wordle solver is to build an optimal guessing strategy. Specifically, a good guess is one that narrows down our search space of potential words as much as possible. We consider this to be the quality of a guess, and we determine the optimal guess at each step using information-theoretic techniques.

[1] https://github.com/AniruddhS24/cuda-wordle

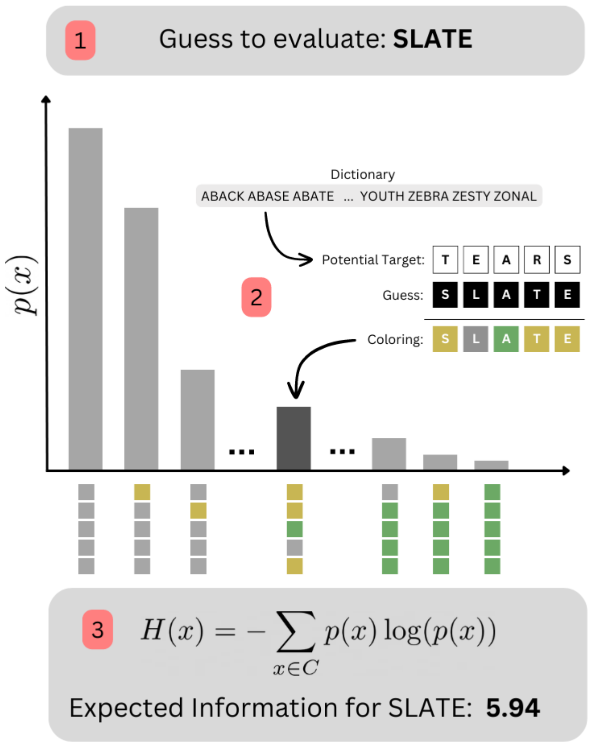

1
Guess to evaluate: SLATE
$p(x)$
Dictionary
ABACK ABASE ABATE ... YOUTH ZEBRA ZESTY ZONAL
Potential Target: T E A R S
2
Guess: S L A T E
Coloring: S L A T E
...
...
3
$H(x) = -\sum_{x\in C} p(x)\log(p(x))$
Expected Information for SLATE: 5.94

Figure 1: Calculating expected information for SLATE. This is done for each word in the dictionary, and the word with the highest expected information is the best guess.

2 Hardware Description

We use an NVIDIA Quadro RTX 6000 GPU with CUDA 11.0. The GPU has 4,608 CUDA cores and 24G RAM. This implies we can run around 4k threads in parallel, which is a massive improvement over the parallelism offered by the machine CPU. Host code was run on an x86 64-core Linux machine (pedagogical-5), which has an Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz.

The GPU memory bandwith up to 672 GB/s, which is especially useful on the nonuniform memory access patterns while updating color distributions. In summary, its highly parallel architecture

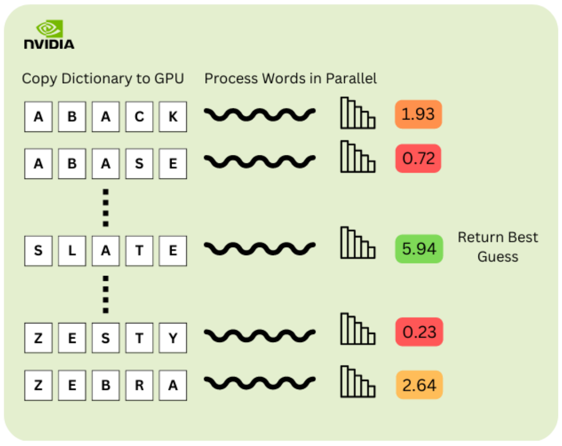


Figure 2: A simplified diagram of our parallelization approach

and fast memory bandwidth enable efficient processing of large dictionaries.

3 Approach

In this section, we detail our approach in computing the expected information of each word as well as how we parallelize operations.

3.1 Expected Information Maximization

To optimilly select guesses, we utalize entropy, a concept in information theory, that relates the unique, high information colorings that significantly cut down the size of the dataset and the probability of such colors occuring in a given word. For our algorithm, we look to maximizing the entropy of the next selected guess conditioned on the remaining dataset, then once a coloring is revealed, update the dictionary based on the information gathered and repeat the process until convergence. More formally,

$$\underset{w \in D}{max}\ \mathbb{E}[I|w] = \underset{w \in D}{max} \sum_{c \in C(w)} p(c) * \log(\frac{1}{p(c)})$$

where $\mathbb{E}[I|w]$ represents the expected information conditioned on guessing on a given word in the dictionary, $C(W)$ represents the set of all colorings for a given word, and $p(c)$ represents the probability of a given coloring for a given dataset. The full sequential algorithm is given in Algorithm 1

3.2 Algorithms

To parallelize the expected information for each word, we spawn CUDA threads to compute the entire coloring distribution over the existing dataset for a given word, thus removing the outer for loop and distributing the word to $|D|$ threads on the GPU. We augment this algorithm using shared memory as well as even more granular multi-threading. In the latter algorithm, we spawn a thread for every word-color pair and use atomicAdd to increment the coloring arrays with the number of words in the dictionary that match the given word-color pair.

Algorithm 1 Sequential Wordle Solver

```
Input: $D$ - Dictionary of potential words
Output: $w^*$ - Optimal guess

$I^* = 0$
for $w \in \mathcal{D}$ do
    $C = []$
    for $g \in \mathcal{D}$ do
        $color = coloring(w|g)$
        $C[color] + +$
    end for
    $I = 0$
    for $c \in \mathcal{C}$ do
        $I = I + \frac{c}{|D|} * log(\frac{|D|}{c})$
    end for
    if $I > I^*$ then
        $w^* = w$
    end if
end for
return $w^*$
```

3.3 Optimization: Coalesced Memory Access

In NVIDIA CUDA-capable GPU architectures, global memory stores and loads are coalesced by the device into as few transactions as possible. More specifically, the concurrent accesses of a warp will coalesce to the number of 32-byte transactions necessary to service all of the threads. On devices of compute capability 6.0 or higher, L1-caching is done by default. If threads within a warp access memory in a scattered pattern, we are likely to make more memory transactions. With large inputs, we may also suffer slowdowns from repeated cache evictions.

Our original (global memory) implementation stored the coloring distributions by concatenating the distribution array for each thread. However, this is poor for coalesced access because threads within a warp access memory locations in disparate locations (i.e. in multiples of $3^5 = 243$, which is the size of each distribution).

A natural solution is to coordinate memory accesses within threads in a warp to be as contiguous as possible. In our implementation, we organize our coloring distribution array to store values for each thread next to each other. As threads accumulate their values for coloring $0, 1, \ldots$ the GPU will make fewer memory transactions as they can be coalesced.

3.4 Optimization: Caching Highly Accessed Colorings in Shared Memory

In GPUs, poor memory access patterns can lead to a variety of performance issues. Shared memory is a powerful optimization because it is on-chip memory, implemented as a partition of the L1 cache. Specifically, this software managed cache is roughly 100x faster than fetching from global memory.

To optimize our code, we wanted to identify irregular memory access patterns in our program. One clear area for improvement is in updating our expected information distributions as we compute colorings across pairs of words. We represent our distributions as contiguous integer arrays in global memory, where each index represents a coloring. However, some colorings are much more probable (across all words) than other colorings. The coloring of all grays, for instance, occurs significantly more often than a coloring with mostly greens. This implies certain memory locations associated with highly probable colorings are memory "hotspots" which are frequently accessed and incremented.

With 5-letter words, we only have $3^5 = 243$ total colorings, so it is feasible to fit the whole array in shared memory. As word length (or block size) increase, however, we are forced to store our distributions in global memory. With 64kB of shared memory, 4-byte integers for coloring counts, the maximal block size is $\frac{64000}{243*4} \approx 65$

This means block sizes greater than 65 will exhaust shared memory space, or choosing a large block sizes forces us to severely limit word length. Increasing block size can be useful for increasing occupancy, and the number of colorings largely limits us. This means we should select a subset of

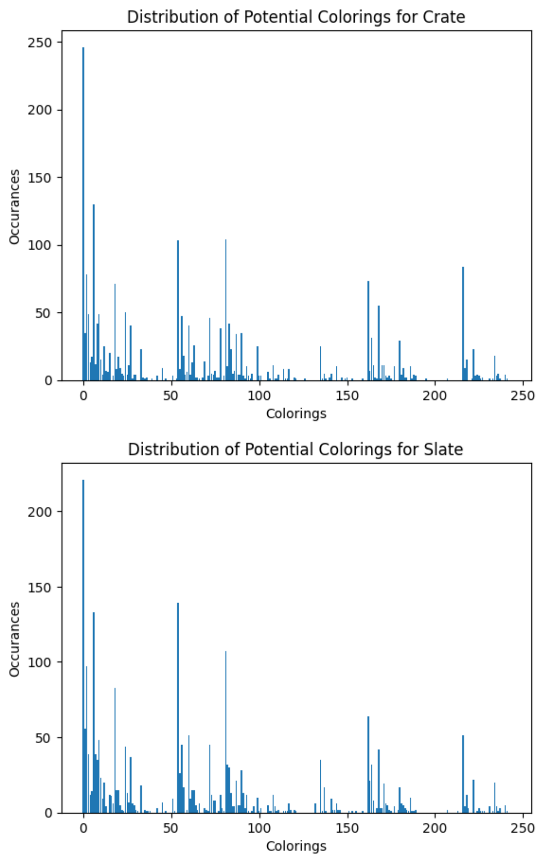


Figure 3: Distributions for CRATE and SLATE on iteration 0

colorings for fast access shared memory, and leave the rest in global memory.

The distributions for most words are heavily right skewed, amassing most of the probability density at the beginning. A natural approach from here is to cache a fixed prefix of the distribution (say 64 colors) in shared memory, while the others remain in global memory. This way, updates to frequent locations (i.e. the all-gray coloring) are guaranteed to be on the L1 cache. The overhead, however, is the added conditional logic to check if a computed color is within the cached prefix or not.

Note the hardware may already be caching some of these values. The issue is GPU cache coherency is ill-defined as the massive amount of parallelism leads to frequent data invalidation. The cache primary tracks information-flow more than frequently used values. By allocating a prefix of the distribution in shared memory, we *ensure* that accesses to certain "hotspot" colorings are fast.

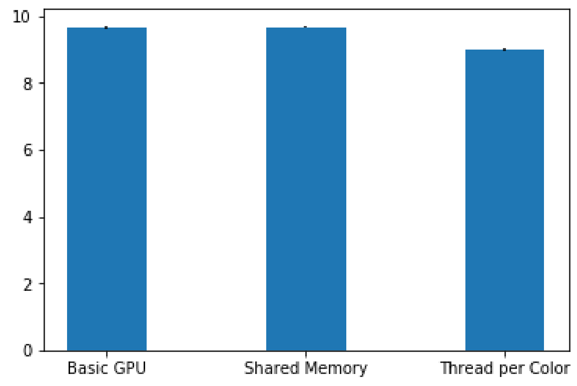


Figure 4: Average speedup across 10 randomly selected words for each implemented algorithm. Average number of iterations for the selected words was 3.5

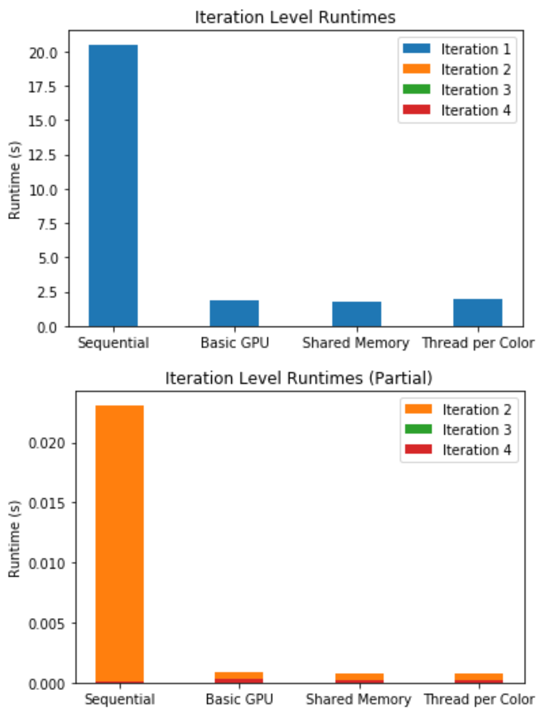


Figure 5: Average iteration level runtime for 10 randomly selected words. Iterations 2, 3 and 4 are shown on the right to visualize their relationships more clearly than is possible in the full graph. No selected word required all 5 iterations to converge to the right answer.

4 Experiments

We implement the information theory based wordle algorithm sequentially and with three distinct CUDA kernels on a corpus of 2315 five letter English words. We benchmark these results on their overall run time along with their ability to scale to different datasets, and the overall throughput of each system.

4.1 End to End Speedup

To determine the impact of each GPU implementation on overall runtime for the given dataset, we randomly sample 10 words from the corpus to use as the target word and determine the average speedup over the sequential algorithm.

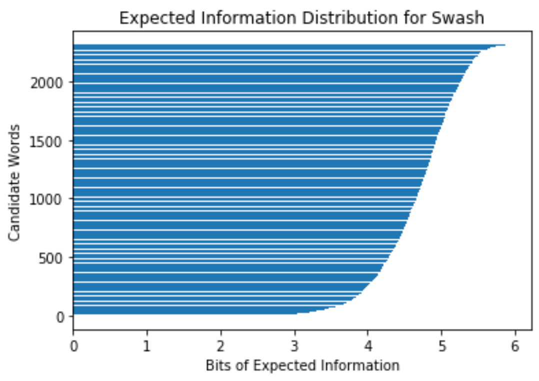
Additionally, we analyze the per iteration runtime to determine how the decreasing dictionary size, that results from updating the corpus from the information of each guess, impacts the overall run time of each algorithm.

In Figure 5 we identified that there is an exponential decrease in the runtime between iterations since the search space of potential words decreases dramatically between iterations of the algorithm. This is further explained by the changing distribution of expected bits of information.

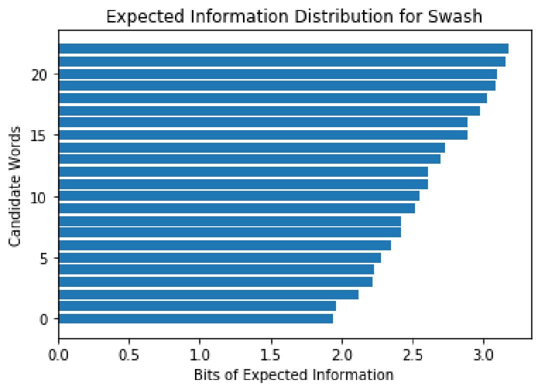
From Figure 6 we can identify how the size of the dictionary changes for a randomly selected word after each iteration. We see that the size of the dataset shrinks exponentially which explains the runtime results seen in Figure 5. We see that not only does the number of calculations necessary decrease from iteration to iteration, but that the new expected information values also decrease, meaning that on average, per iteration, less words are removed from the dictionary which further allows later iterations to perform better than earlier ones.

It is important to note however that as the number of words in the dictionary decrease, the throughput of the sequential algorithm increases, making it more performant in later iterations. As seen in Figure 7 after the first iteration the sequential algorithm has better throughput compared to all the GPU based algorithms.

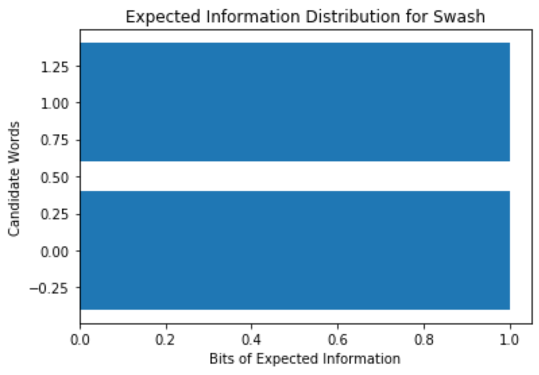
Finally, after modeling the end to end speedups of all three GPU algorithms, we notice that each of the implemented algorithms garnered a significant speedup from the sequential (Figure 4), however, when comparing the shared memory implementation and the thread per color implementation, we notice that there is no significant speedup from the basic GPU implementation. This lets us conclude that the necessary coordination for the GPU algorithms do not out weight the accelerated computation for the implemented algorithm for the basic wordle dataset. In the following sections, we explore how these relationships change as we scale to various other datasets and explore why the shared



(a) Initial information distribution: Guess slate



(b) Information distribution on second iteration: Guess shark



(c) Information distribution on third iteration: Guess swash

Figure 6: Expected information distribution over solver iterations. Three guesses were required before getting the required answer for the selected word swash

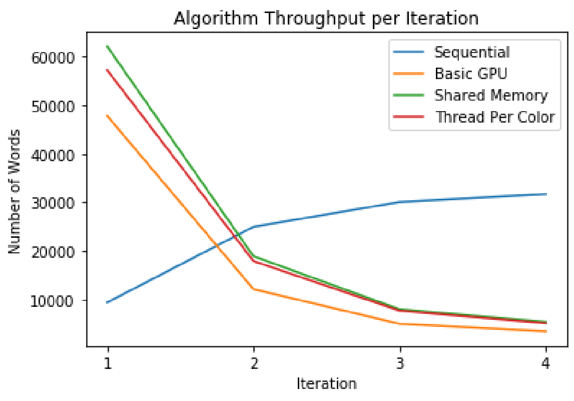


Figure 7: Average throughput for 4 algorithms across 10 randomly selected words for each implemented algorithm. Average number of iterations for the selected words was 3.7

memory and thread per color GPU optimizations do not yield significant performance improvements over the basic implementation.

4.2 Sentence Level Decoding

Expanding on the basic wordle interface, we implement a sentence level guesser by tokenizing a larger vocabulary of words and leverging the same expected information algorithm to see how the various techniques adapt to a larger corpus of potential answers as well as a larger vocabulary. For this task, we use the Stanford Natural Language Inference Corpus. We build two distinct sentence datasets to benchmark the various algorithms. The first *small* dataset contains 273 distinct 10 word English sentences and is composed from a vocabulary of 820 unique English words. The second *large* dataset contains 2474 distinct 10 word English sentence that is composed from a vocabulary of 3096 words. An important note is that we downsized the large corpus from 6500 sentences with a vocabulary of 8000 words since the sequential algorithm was unable to finish in a reasonable amount of time (30+ minutes) while the GPU accelerated algorithms were all able to finish in less than a minute.

Dataset	Small	Large
Basic GPU	0.929	13.745
Shared Memory	0.973	13.982
Threads per Color	0.256	11.748

Table 1: Algorithm runtimes on various dataset

On the small dataset, all three GPU algorithms performed slower than the sequential baseline. This demonstrates the necessary tradeoffs between corpus size and the use of GPU coordination. Contrasting this to the large dataset, however, we can see in Table 1 that as the corpus size increases, so does the efficiency of the GPU algorithms. Additionally, notice the distinct lowered speedup in the threads per color algorithm compared to the basic GPU and shared memory implementations. We believe that due to the distribution of colorings, the thread per color algorithm leads to a significant percentage of the threads being serialized which leads to the worse speedup. This hypothesis is explored and confirmed in Section 4.6.

4.3 Scaling Words

To further test the effectiveness of the various algorithms in different contexts, we generate custom set of inputs. For this task, we uniformly select characters of the English alphabet to form 5 letter "words", similar to the basic wordle structure.

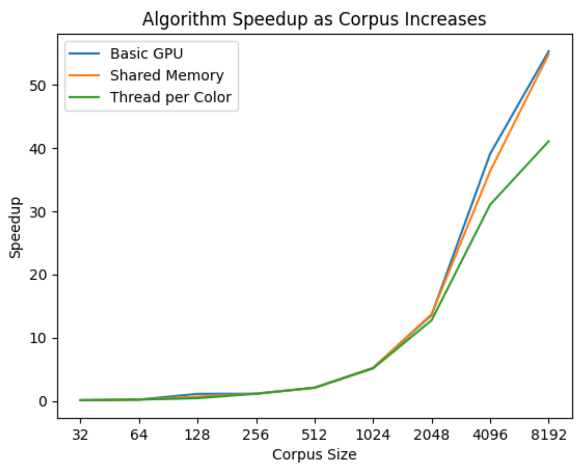


Figure 8: Change in speedup for various algorithms as corpus size changes

We noticed a strong trend between speedup and corpus size. As we increased the number of potential words, the speedup across all three algorithms increased significantly, reaching a near 60x speedup when only 8000 words where present in the corpus. This is likely because the number of scheduled threads on the GPU is directly proportional to the number of words in the corpus since each thread find the information distribution for a given word.

4.4 Scaling Vocabulary Size

We implement a similar experiment to scaling the size of the vocabulary to visualize any relationships between the GPU algorithms and the number of potential guesses for a given position in the word.

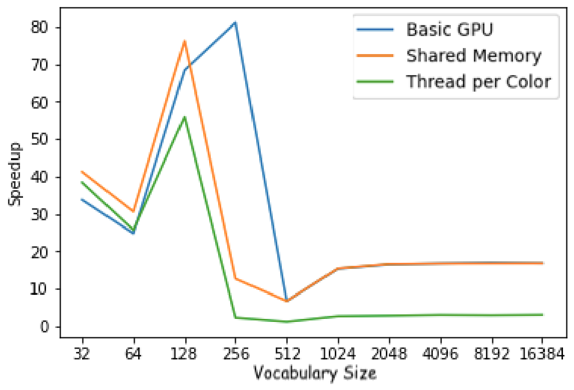


Figure 9: Change in speedup for various algorithms as vocabulary size changes

While there is an initial trend of increasing speedup, we notice that the basic and shared memory implementation converge to an 20x speedup while the thread per color algorithm converges to 5x. This is likely because the vocabularly size itself doesn't directly influence the number of spawned threads which would lead to a stagnation in the speedup growth.

4.5 Scaling Number of Letters

Finally, we wanted to identify if there was any significant impact of scaling the number of letters in the target word on overall speedup.

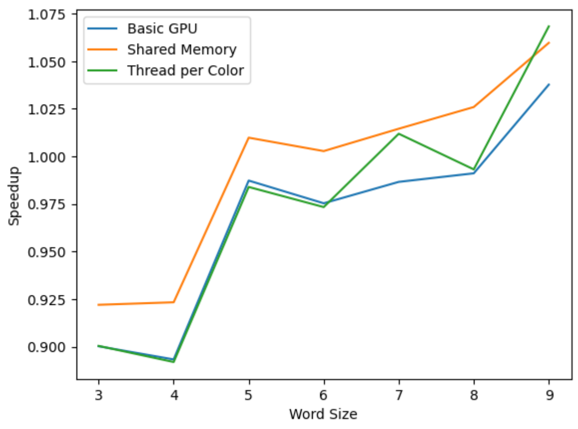


Figure 10: Change in speedup for various algorithms as word size changes

We notice a very slight growth in speedup across all three algorithms. This is likely because when computing the colorings, there is more information that needs to be processed for each pair of words which is accelerated on the GPU system. While this holds for smaller word lengths (< 10), we hypothesize that as the word length increases, the memory demands that increase exponentially will be more dominant and slow down the GPU algorithms when benchmarked against the sequential ran on the CPU.

4.6 Memory Access Patterns

In Section 3.2 we introduced a memory optimization that resulted from irregular memory access patterns. By using the distribution shape to cache frequently accessed colors, we attempted to speed up our expected information calculations. In this section, we actually plot the memory access patterns on the distribution array to verify our hypothesis. That is, if CUDA is storing color distributions as contiguous blocks of memory, the memory access pattern should mimic our empirical distributions.

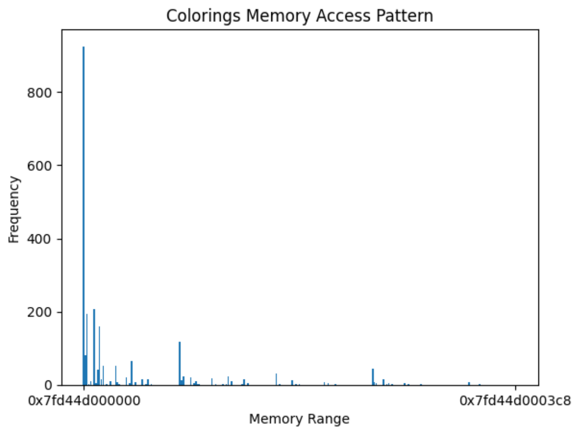


Figure 11: Memory access for a word during the first guess (Iteration 0)

The first 64 entries (256 bytes) account for 89.7% of total memory accesses during this step. Assuming shared memory latency is 100x lower than global memory, and assuming each global memory access takes the same amount of time, the optimized algorithm should roughly take $(0.9) * (1/100) + 0.1 \approx 0.11$ or 11% of the original global memory execution time. Note this is a very rough estimate, as we do not factor in automatic caching or per-request latency variance.

We compare two implementations: using global memory for the whole distribution, and caching a prefix in shared memory (optimization). The results indicate a small but valuable speedup, which will become more realizable if we scale our system (i.e. sentence-based Wordle).

5 Conclusion

After experimenting with various algorithms across a rich range of datasets, we were able to conclude that GPU boosted wordle is effective at significantly speeding up the runtime of the wordle algorithm. We saw an 10x speedup in the motivating example of the Wordle corpus and demonstrated the increasing performance as the dataset scales. We identified that due to the distribution of memory accesses, further levels of multi-threading based on thread-color pairs were not effective since a significant portion of threads were searialized, making the creation and coordinitation overheads not worth while. In the future, we hope to explore more advanced structures for decoding, using prior language distributions to for guesses without access to the corpus of potential words, and further analyze occupancy metrics accross the various algorithms.

Appendix

This project took us around 50 hours to complete.