SOLVING THE WORD PROBLEM ON CUDA

SAYANTAN KHAN

Contents

1. The word problem for hyperbolic groups

A finitely presented group is a group described by two finite pieces of data, the generators and the relations. The generators are a finite set of letters $G = \{g_1, \ldots, g_n\}$ such that any group element can be expressed as a product of the g_i's or their inverses. The relations are a finite set $R = \{r_1, \ldots, r_m\}$ of words in the generators and their inverses, and these words evaluate to the identity element in the group. A group Γ can be defined uniquely by specifying the two finite sets $\langle G \mid R \rangle$.

However, given such a presentation, to actually perform calculations with the group elements, it is important to know whether two different words in the generating set correspond to the same group element. For instance, consider the group presented in the following manner: $\Gamma = \langle \{a, b\} \mid \{aba^{-1}b^{-1}\} \rangle$. In this group, the words ab and ba correspond to the same group element. This is what is known as the word problem for finitely presented groups.

Question 1 (Word problem). Given a finitely presented group $\Gamma = \langle G \mid R \rangle$, and a pair of words (w_1, w_2) in the generators, do they correspond to the same group element in Γ?

With a presentation, we can turn the group Γ into a metric graph: the vertices correspond to elements of the group, and there is an edge between two vertices γ_1 and γ_2 if there is some $g \in G$ such that $\gamma_2 = \gamma_1 \cdot g$ or $\gamma_2 = \gamma_1 \cdot g^{-1}$. We set each edge to have length 1, and we set the distance between two points γ_1 and γ_2 to be the length of the shortest path between γ_1 and γ_2. The shortest path between two points is called a *geodesic* between the two points. This lets us pose the following question about finitely presented groups.

Question 2 (Geodesic representative). Given a finitely group $\Gamma = \langle G \mid R \rangle$, and a word w, what is a geodesic representative of w, i.e. a word of minimal length equivalent to w?

Note that the geodesic representative of a word may not be unique, and if we want a unique geodesic representative for a word, we put an arbitrary ordering on the generators, and look for lexicographically smallest geodesic representative: we call this a ShortLex representative.

Question 3 (ShortLex representative). Given a finitely group $\Gamma = \langle G \mid R \rangle$, and a word w, what is the ShortLex representative of w, i.e. the lexicographically smallest word of minimal length equivalent to w?

Observe that having an algorithm to solve the geodesic representative or ShortLex representative problem automatically gives us an algorithm to solve the word problem since the geodesic/ShortLex representative for $w_1 w_2^{-1}$ must have length 0 if $w_1 = w_2$ in Γ.

For arbitrary finitely presented groups, it was shown in 1955 that the word problem is undecidable, which also means Questions 2 and 3 are also undecidable for arbitrary finitely presented groups. That means if we want algorithms to answer the above questions, we need to impose additional hypotheses on the finitely presented groups we are looking at.

The condition we will impose is *hyperbolicity*: a group is δ-hyperbolic (where δ is a positive integer) if for any geodesic triangle, all the edges are within distance δ of the other two edges, no matter how large the triangle. The

Date: January 2022.

simplest example of a hyperbolic group is the following group $F_2 \coloneqq \langle \{a, b\} \mid \{\} \rangle$. The metric graph associated to this presentation is a 4-valent tree, and it's easy to see that this group is 0-hyperbolic.

For hyperbolic groups, the geodesic/ShortLex representative problems are indeed decidable, and in fact there exists an $O(n^2)$ serial time algorithm to find geodesic representative of a given word of length n, where the constant depends on the group.

In [1], Cai proved that computing the geodesic representative was in the complexity class NC^2, namely, given polynomially many processors, the geodesic representative for a word of length n can be computed in $O((\log n)^2)$ time. In the following sections, we adapt Cai's algorithm to compute the ShortLex representative, and show that the ShortLex representative problem can be solved in $O((\log n)^2)$ time, although in practice it has much better constants, and therefore is much faster, at the expense of requiring $O(n^3)$ processors as opposed to $O(n^2)$ as in the geodesic case. We also compute the space complexity of the algorithms, and implement the ShortLex version in CUDA.

2. A PRAM algorithm

Our model for a PRAM in this section will be a *Concurrent Read Concurrent Write* PRAM, and there will be a few instances where several processors will attempt to write on the same memory location. In those situations, we will describe precisely how the concurrent writes take place.

A naïve first attempt at parallelizing the serial algorithm to get a parallel algorithm might go something like the following. Given a word w, break it up into two equal sized chunks $w = w_1 w_2$, and compute the geodesic representatives $\overline{w_1}$ and $\overline{w_2}$ of w_1 and w_2 respectively and concatenate them. However, the concatenation of two geodesics need not be a geodesic, so we still need to compute the geodesic representative $\overline{\overline{w_1} \cdot \overline{w_2}}$ of $\overline{w_1} \cdot \overline{w_2}$. Note that the three geodesics $\overline{w_1}$, $\overline{w_2}$, and $\overline{\overline{w_1} \cdot \overline{w_2}}$ form a geodesic triangle, so we might be able to reduce the amount of computation required by using the geometry of the group, namely hyperbolicity (recall that hyperbolicity is a uniform estimate about triangles, independent of their sizes).

But before we can speed up the concatenation of geodesics, we need to speed up another primitive operation, namely multiplying a geodesic word w with a generator on the right, i.e. computing the geodesic representative $\overline{wg}$ of wg. The serial algorithm for this operation takes $O(n)$ time, and this operation is used several times in the divide-and-conquer, which means the first step in speeding up the divide and conquer is speeding up right multiplication by a generator. This is the step we will first parallelize.

Step 1: Parallelizing right multiplication by generator. Given a geodesic/ShortLex word w of length n, and a generator g, this algorithm will compute the geodesic/ShortLex word for $w \cdot g$ in $O(\log n)$ time provided we are given $O(n)$ processors.

We start with a finite state automaton M_g: this automaton over the alphabet $\{G \cup \{\phi\}\} \times \{G \cup \{\phi\}\}$. It accepts a pair of words (w, w') (where the shorter word is padded by ϕ to have the same length as the longer word) if $w' = w \cdot g$ and both w and w' are geodesic/ShortLex words. This automaton only depends on the group Γ and the generator g, so this can be pre-computed for the finitely many generators, and stored in memory. Given a geodesic/ShortLex word w, and a generator g, our algorithm will compute the word w' such that M_g accepts (w, w') (see [2] for more details on the automaton M_g).

First, we initialize an $s \times s \times (n+1)$ with some placeholder value, say b. Here, s is the number of states of the automaton M_g, which is bounded above by some constant depending on the group, which means we can do this operation in $O(1)$ time using $O(n)$ processors. At coordinate (s_i, s_f, k) of the matrix, we store a letter l if the automaton M_g transitions from state s_i to s_f upon reading $(w[k], l)$, where $w[k]$ is the k^{th} letter of w. Observe that this step can again be done in $O(1)$ time using $O(n)$ processors, since we have the automaton M_g stored in memory, and the number of states and generators is a constant only depending upon the group Γ. Observe that if the coordinate (s_i, s_f, k) still has the placeholder value b, that indicates there is no letter l such that the automaton goes from s_i to s_f upon reading $(w[k], l)$: the placeholder values therefore serve as a proof of absence of paths.

The next step is to combine these length one paths into longer and longer paths, until we have a length $n+1$ path between every pair of states. Then the path starting at the initial state of the automaton and ending at any of the accepting states will be a word w' such that M_g accepts (w, w'). We perform this combination of paths in an inductive manner. We first assume without loss of generality that the length $n+1$ is actually a power of 2, i.e. $n + 1 = 2^m$. We can assume this by padding w with enough padding symbols to make its length a power of two; this makes it at most twice as long as it initially was. Suppose now at step j we have paths of length 2^j: we do the following in parallel over all $1 \leq p \leq 2^{j-1}$ and for all triples of states (s_i, s_m, s_f). We check if there is a path of length 2^j at coordinate $(s_i, s_m, 2^{j+1}p)$: recall that this path will be a word v such that the automaton

will go from s_i to s_m upon reading $(w[2^{j+1}p \dots 2^{j+1}p + 2^j], v)$[1]. We then check if there is a path of length 2^j at coordinate $(s_m, s_f, 2^{j+1}p + 2^j)$. If there are paths at both of those coordinates, then concatenating them will clearly give a path of length 2^{j+1} from s_i to s_f. We write such a path on a new matrix of the same dimensions at coordinate $(s_i, s_f, 2^{j+1}p)$. Note that the only occasion where this algorithm might try to concurrently write to the same location is that if for different values of s_m, it finds paths from s_i to s_m and s_m to s_f. We get around this issue by acquiring a lock on the location we're writing to so that only one group of processors can be writing to the location at the same time. This means our PRAM model needs some mechanism of locking, for instance, atomic variables.

We now analyze the time complexity of the above operation: we need $O(n)$ processors and $O(1)$ time. Note that the concatenation operation could take $O(n)$ time if done naïvely, but we do that in parallel as well. Finally, note that to get a path of length 2^m, we will need to do the above operation m times, giving us a time complexity of $O(m) = O(\log n)$, and we will need $O(n)$ processors. For practical analysis later, we compute the exact number of processors needed: at any given stage, we spawn at most $s^3 n$ processors.

Observe that we can also use the above algorithm to right multiply with words of arbitrary length: doing so for a word of length n will require $O(n \log n)$ time, which is already an improvement over the serial algorithm, which requires $O(n^2)$ time. For the next step, we will need to multiply with words of length at most 2δ, where δ is the hyperbolicity constant. Since that quantity is fixed for a given group, the time complexity is still $O(\log n)$, but with slightly worse constants.

Step 2: Parallelizing concatenation. Before we outline the algorithm to parallelize concatenation, we will need two lemmas. The proof of these lemmas is easy in the case of geodesics, but requires some work in the ShortLex case. We will skip the proof of these lemmas since that will require a digression into the theory of hyperbolic groups.

Lemma 1. *Given a geodesic/ShortLex representative of a word, any substring of this word is also geodesic/ShortLex.*

Lemma 2. *Given a geodesic/ShortLex representative of a word, its inverse is also geodesic/ShortLex.*

The parallel algorithm for concatenation relies on the following observation about hyperbolic groups. Suppose we are given two geodesic/ShortLex words w_1 and w_2 of length at most n. Let $\overline{w_1 w_2}$ be the geodesic/ShortLex representative of their product. Then the three words w_1, w_2, and $\overline{w_1 w_2}$ fit in the geodesic triangle depicted in Figure 1.

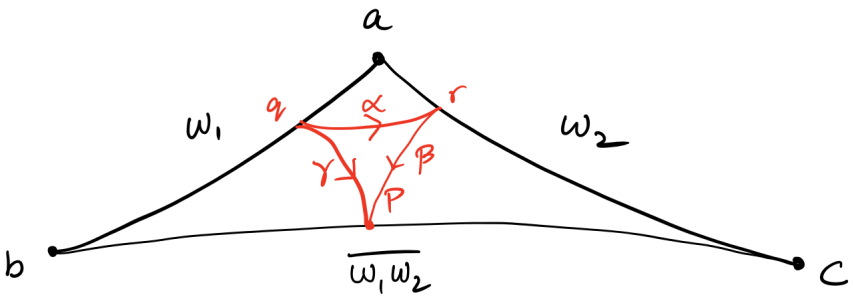

FIGURE 1. Geodesic triangle formed by w_1, w_2 and $\overline{w_1 w_2}$.

From hyperbolicity, we know that there are points p, q, and r on $\overline{w_1 w_2}$, w_1 and w_2 such that the distance between q and p is at most δ, and the distance between p and r is also at most δ, and by the triangle inequality, the distance between q and r is at most 2δ. The edges α, β, and γ form a triangle with edge lengths at most 2δ, δ, and δ. We will call such a triangle a *small triangle*. Note that for any given group Γ, there are only finitely many small triangles: we pre-compute all small triangles for the group and store them in memory.

We can now describe the geodesic concatenation algorithm. For all points q on w_1, and all small triangles, we check in parallel if $aq \cdot \alpha$ is a substring of w_2. Note that we can multiply the segment aq with α using Step 1, since aq is also a geodesic/ShortLex by Lemmas 1 and 2, and α has length at most 2δ. That means this operation takes $O(\log n)$ time and $O(n^2)$ processors, since we are iterating over q in parallel, which requires n processors, and each processor is calling the multiplication function, which requires $O(n)$ processors again.

If $aq \cdot \alpha$ is indeed a substring of w_2, we denote the endpoint of that substring by r. We then compute the geodesic/ShortLex representatives of $bq \cdot \gamma$ and $cr \cdot \beta$. This operation again takes $O(\log n)$ time since β and γ have

[1]The word $w[i \dots j]$ refers to the subword of w from index i to index j.

length at most δ. We store in memory the concatenation of $\overline{bq \cdot \gamma}$ and $\overline{(cr \cdot \beta)^{-1}}$, as well as the sum of their lengths. We must have that the concatenation with the shortest length is the geodesic representative. We then find the index of a minimal length concatenation using the standard $O(1)$ time algorithm for finding the minimum. Recall that this algorithm uses $O(n^2)$ processors, which means in this scenario, the maximum number of processors used will be t^2n^2, where t is the number of small triangles.

However, the above algorithm for finding the minimal length concatenation is not enough to find the ShortLex representative, since the ShortLex representative is the minimal length word which is also lexicographically the shortest.

Step 3: Finding the smallest ShortLex word. First, we describe the algorithm to compare two words w_1 and w_2 of length at most n. If w_1 and w_2 have different lengths, the algorithm returns the shorter of the two words. If the words have length 1, the algorithm returns the lexicographically smaller word in constant time. If the words have equal length more than 1, the algorithm splits them up into two chunks of approximately equal size w_1' and w_1'', and w_2' and w_2''. It then recursively compares w_1' and w_2' and w_1'' and w_2'' in parallel. If w_1' is smaller, the algorithm returns w_1, and if w_2' is smaller, the algorithm returns w_2. If the first chunks are equal, then the result depends on the comparison between w_1'' and w_2''. This comparison algorithm takes $O(n)$ processors and $O(\log n)$ time.

Now that we have a way of comparing words lexicographically, we use the same algorithm for computing minimum in parallel, which will take $O(\log n)$ time and use $O(n^3)$ processors.

Step 4: Putting it all together. We now have all the subroutines we need to find the geodesic/ShortLex representative of a word of length n. If the word is of length 1, it is already its own geodesic/ShortLex representative so there is nothing to do. If the word is longer, we split it into two halves of roughly equal size, and compute the geodesic/ShortLex representative in parallel. Suppose this takes $f\left(\frac{n}{2}\right)$ time. Concatenating the two geodesic/ShortLex words takes an additional $O(\log n)$ time, which gives us the following recurrence for f.

$$f(n) \leq f\left(\frac{n}{2}\right) + c\log(n)$$

This shows that the runtime f is $O((\log n)^2)$, and the maximum number of processors required is t^2n^2 and t^2n^3 in the geodesic and ShortLex cases respectively.

3. A CUDA implementation of ShortLex

In this section, we outline why we chose to implement this PRAM algorithm on CUDA, as well as describe some of our design decisions.

Firstly, we picked CUDA rather than OpenMP because of the large number of processors required: since the number of processors required is t^2n^3, and our values of n ranged from 10 to 3000, the number of CPU processors would not be enough to observe actual speedups. It is for the same reason we did not pick OpenMPI either, because while this algorithm is easy to adapt to a distributed memory system, even a distributed memory system will not have enough cores to satisfy the scaling requirements of the algorithm.

However, a distributed GPU version of this algorithm could work quite well: given 1000s of machines, each with 10000 GPU cores, we would have access to millions of processing units, and the communication overhead would be low, since one could have a linear network, and a node would only ever need to communicate with its immediate neighbours, and only transmit $O(n)$ data.

The first step in implementing this algorithm was to deal with the serial computation that only needed to be done once for a given group, namely computing the automatons M_g, and the set T of small triangles. Rather than implementing this ourselves, we chose to go with the existing state-of-art algorithm, namely the Knuth-Bendixon completion algorithm. The GAP packages kbmag[2] has a fairly performant implementation that we used as input for our program.

The next step was to implement the right multiplication algorithm. Rather than instantiating an $s \times s \times n$ matrix for each of the algorithm, we chose to instantiate two such matrices at the very beginning, and use them in an alternating manner. This was mainly for performance reasons, as allocating memory is time consuming, especially when allocating memory from within a GPU kernel. Whenever possible, we chose to allocate a large chunk of memory at the very beginning, and manually manage that memory from the kernels.

Another performance improvement we observed was that when copying buffers within a kernel, it's better to copy them manually using a `for` loop rather than using `memcpy`.

[2] `https://www.gap-system.org/Packages/kbmag.html`

The biggest challenge was avoiding warp divergence: one decision we needed to make was when spawning a large number of threads, whether the threads in a single block/warp should correspond to the same s_i, s_f, or k. We ultimately decided to go with the last choice, namely adjacent threads correspond to adjacent values of k, since that resulted in adjacent threads accessing adjacent regions of memory, which improved performance. However, this led to branching within the same warp, but that wasn't a significant bottleneck since only one of the branches had a lot of computation.

We also implemented the function to multiply with words of arbitrary length. One design decision we made here was to minimize the number of allocations: rather than allocating an $s \times s \times n$ matrix each step, we allocated two of them at the very beginning, and reused them over and over again. This function also gives us a $O(n \log n)$ time algorithm for geodesic/ShortLex representative.

When implementing the geodesic concatenator, rather than the base case of the induction being length 1 words, we chose to let the base case be words of length 3δ, using the function described earlier. We expect that this will improve performance slightly, although we were unable to verify it (see section 4). In any case, one will need to tune this parameter for optimal performance.

We also chose to make this algorithm iterative rather than recursive, because of the associated overhead with recursive algorithms.

One limitation of the current implementation is that we synchronize at each step: rather than continuing to concatenate whenever adjacent terms have been computed. We tried to avoid this excessive synchronization, but that resulted in strange and hard to debug race conditions.

4. Practical limitations

To benchmark our implementation of the algorithm, we start with the two simplest non-trivial hyperbolic groups: the surface group S_2 of genus 2 and the Coxeter 334 group C_{334}. In particular, we use the following presentations for the above 2 groups.

$$S_2 = \left\langle \{a, b, c, d\} \mid \{aba^{-1}b^{-1}cdc^{-1}d^{-1}\} \right\rangle$$
$$C_{334} = \left\langle \{a, b\} \mid \{a^3, b^3, (ab)^4\} \right\rangle$$

The number of states in M_g for the group S_2 is 105, and the number of states for C_{334} is 78. The hyperbolicity constants for the two groups are both 5: this lets us compute the number of small triangles in both of these groups. For S_2, there are 496799521 small triangles, and for C_{334} there are 11881 small triangles. Despite all these parameters being constants, they are somewhat large for the hardware we have access to.

For instance, storing the database of large triangles would take almost 38 gigabytes for S_2 and almost 1 gigabyte for C_{334}. While the latter database can be stored in the GPU memory, we will need to store for each element in the database, the pair $bq \cdot \gamma$ and $cr \cdot \beta$, which together will exceed the available memory even for small values of n. This is the reason why we were unable to test Step 2 of the algorithm on large input sizes.

The above is not the only limitation though. To achieve an $O(\log n)$ run time for step 1, we need to spawn $s^3 n$ processors simultaneously: in our examples the values of s are 105 and 78 respectively, which means we need to spawn $105^3 n$ and $78^3 n$ processors. Since that is way more than the number of simultaneous threads that can run on current GPUs (they can handle about 5000 simultaneous threads), we make a change to the algorithm. Rather than iterating over s_m in parallel, we iterate over it serially, reducing the number of processors required to $s^2 n$, at the expense of the runtime getting multiplied by s.

While this compromise still results in more threads than number of processors, it is faster, though the run time scales linearly with the input size rather than logarithmically, since only a small chunk of the threads are running at any given time. However, we do see that the runtime scales appropriately as we change the number of threads, as one would expect (see section 5).

Given the limitations of this implementation are essentially due to the constraints in the available memory and maximum number of threads, this might be be a practical implementation in the future, when GPUs with significantly more memory and threads are available. Also, recall that this algorithm can be adapted easily to a distributed memory model with minimal connections, which means that the one can get around the maximum number of thread limitations currently, by adding more GPUs to the cluster. The memory requirements however, are a real bottleneck, and there seems to be no way of getting around them other than increasing the available memory significantly.

5. Analysis of timing results

Before we analyse the timing data, we describe how the number of parallel processors in the PRAM algorithm is mapped to parallelism in the actual code. At any stage, if the PRAM algorithm calls for parallel execution

on p processors, the code calls a function on b blocks, each block containing t threads, where $bt \approx p$. While a block is running, all the threads within the block execute simultaneously, it is not true that all blocks execute simultaneously. The GPU we ran our code on was an NVidia V100, which can support 80 streaming multiprocessors executing simultaneously. The number of blocks that can run on a streaming multiprocessor simultaneously depend on a number of tunable factors, of which we will only tune one, namely the number of threads per block. Since the number of blocks per streaming multiprocessor is 16, and we have 80 streaming multiprocessors, scaling the block size should scale the number of concurrent threads in a proportional manner.

We will range the number of threads per block from 1 to 1024, and the following table lists how many active threads across the entire GPU those values correspond to. These calculations are not exact, since the number of active threads also depends on a host of other parameters we are not monitoring, like memory usage within a block, warp-divergence, etc.

Threads/Block	Blocks/SM	Total threads
1	16	40
2	16	80
4	16	160
8	16	320
16	16	640
32	16	1280
64	16	2560
128	12	1536
256	6	1536
512	3	1536
1024	1	1024

TABLE 1. Number of active threads as we vary block size.

Note that the maximum number of threads running in parallel is 2560; on the other hand, to achieve the logarithmic scaling in our PRAM algorithm, we need orders of magnitude more processors running simultaneously. This means we will see a linear scaling, but we should still be able to observe a speedup as we increase the number of threads (i.e. block size).

First, we plot the runtime of the right multiplication function as a function of input size for some fixed values of block size (see Figure 2 and Figure 3).

As we expected, the runtime scales linearly with input size, and increasing the block size reduces the runtime across a range of input sizes. However, the fastest runtime is not when the block size is 64, as the table would suggest, which means 64 onwards, some other factor is the bottleneck.

To observe this better, we plot speedup vs. block size for a single input size (see Figure 4 and Figure 5).

Observe that in both of the scaling plots, we have faster scaling until block size 64 or 128, and then the scaling slows down, which might be possibly because of some other bottleneck at large block sizes. The above results show that the right multiplication function scales fairly well. Next, we see how well the ShortLex representative function scales.

First, we plot the runtime vs input size for various block sizes (see Figure 6 and Figure 7).

Here, we see that the runtime is superlinear in the input size: since the previous step was $O(n)$ rather than $O(\log n)$ (because we cannot run arbitrarily many processors in parallel on the hardware), the ShortLex representative algorithm will take $O(n^2)$ time, as is evident in the plots. We also notice that this function does not scale as well as the previous one: increasing the block size does not improve performance significantly. We analyze them more carefully using speedup plots (see Figure 8 and Figure 9).

Clearly, we are getting nowhere close to the speedups we saw for the right multiplication function. Analyzing the program using `nvprof` reveals the reason for this slowdown: a very large fraction of the time of the program's runtime is spent clearing the entries of the matrix used to compute paths. In principle, since the matrix has $O(n)$ entries, we should be able to reset its entries in $O(1)$ time; however `cudaMemset` is not as efficient as one would expect, and thus takes $O(n)$ time to reset the entries of the matrix.

One possible way to speed up `cudaMemset` would be to write a parallel version of it, that launches multiple kernels, and resets the memory in parallel in between calls to the right multiplication function. We chose to not implement that as we were running short on time, but that fix will almost certainly improve the speedup of the ShortLex representative function.

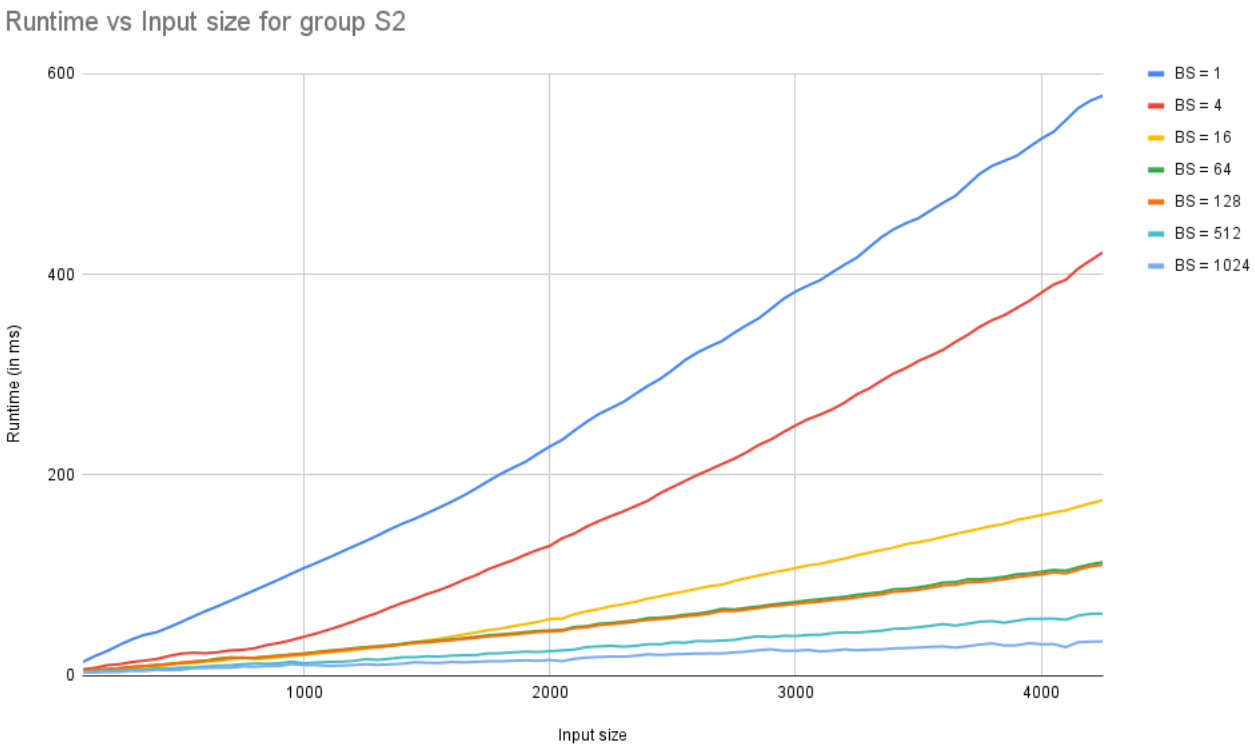

FIGURE 2. Runtime vs input size for S_2.

REFERENCES

[1] CAI, J.-Y. Parallel computation over hyperbolic groups. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of Computing* (1992), pp. 106–115.

[2] EPSTEIN, J. C. D. B., AND HOLT, D. Word processing in groups. *Bull. Amer. Math. Soc 31* (1994), 86–91.

Email address: saykhan@umich.edu

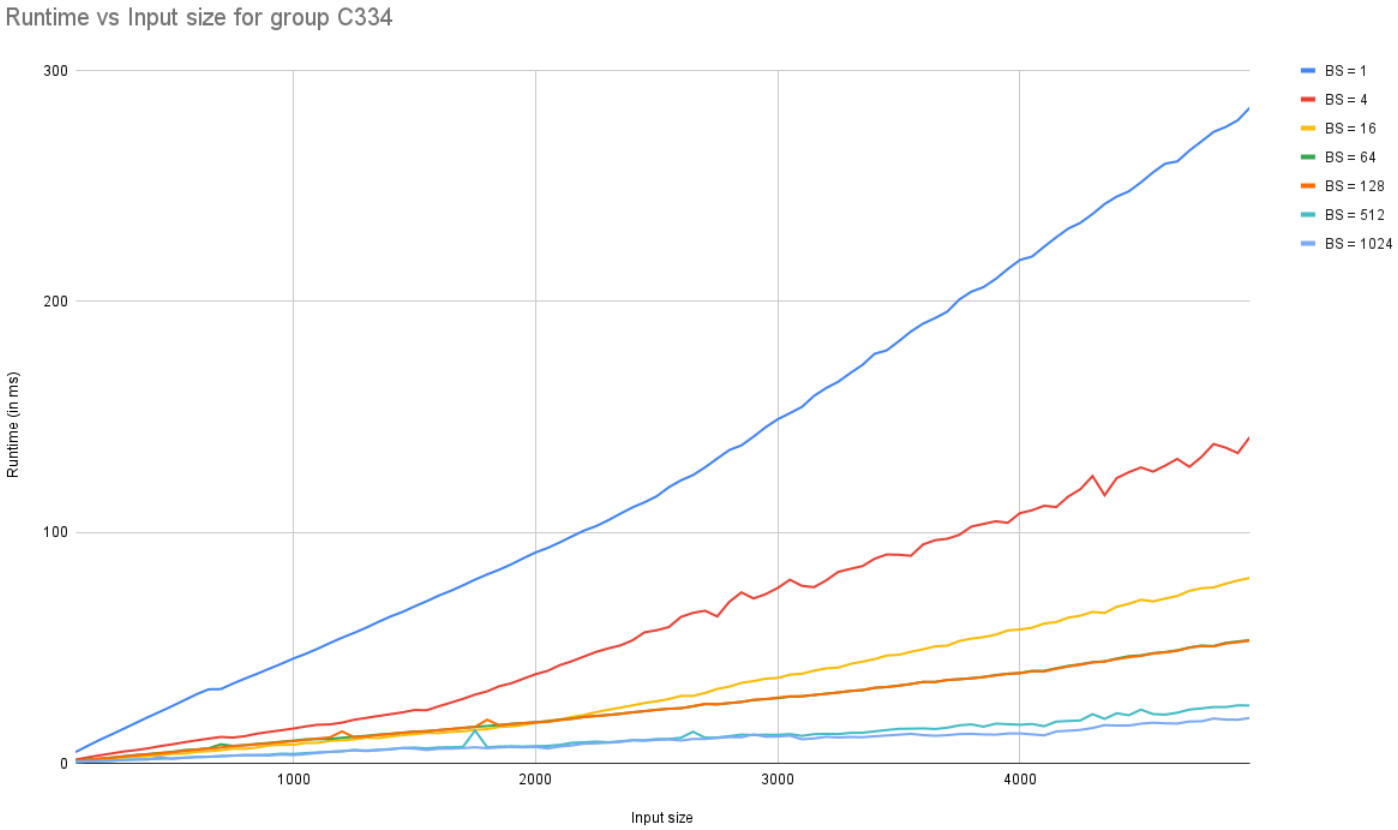

FIGURE 3. Runtime vs input size for C_{334}.

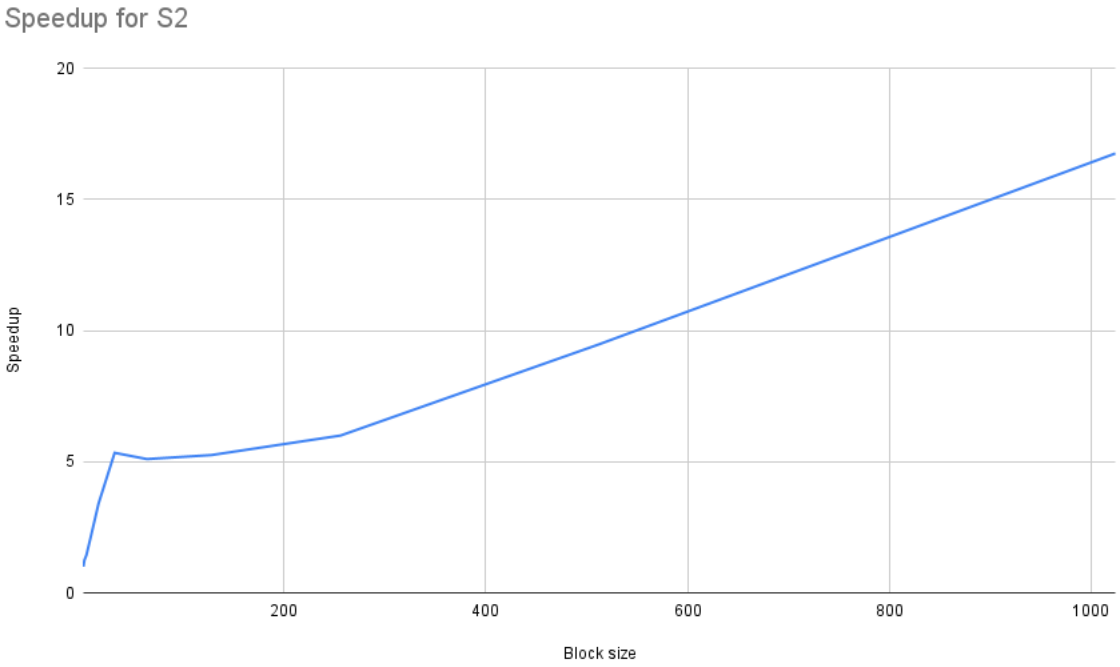

FIGURE 4. Speedup vs block size for S_2.

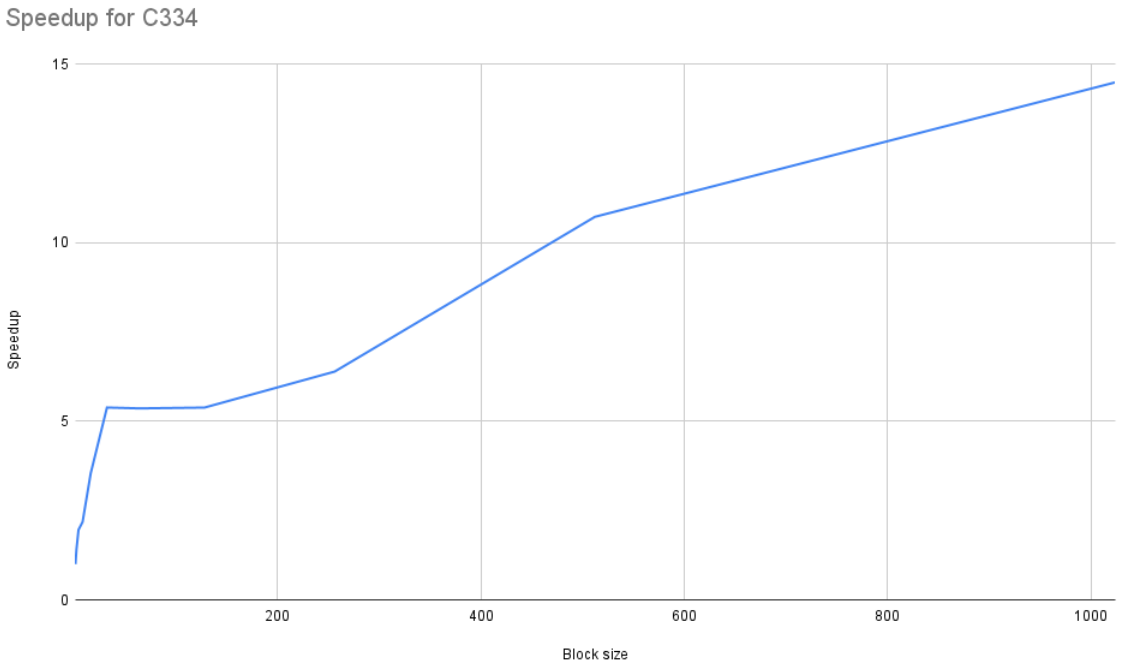

FIGURE 5. Speedup vs block size for C_{334}.

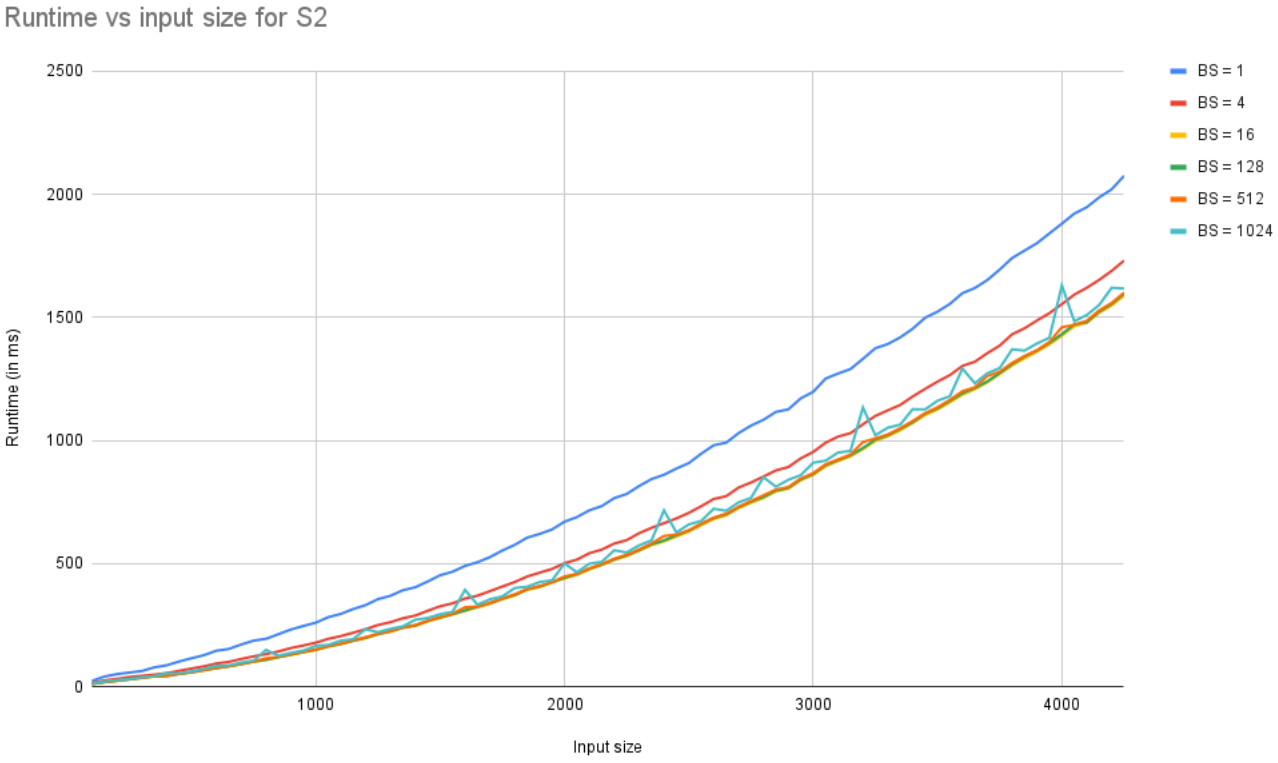

FIGURE 6. Runtime vs input for for ShortLex representative in S_2.

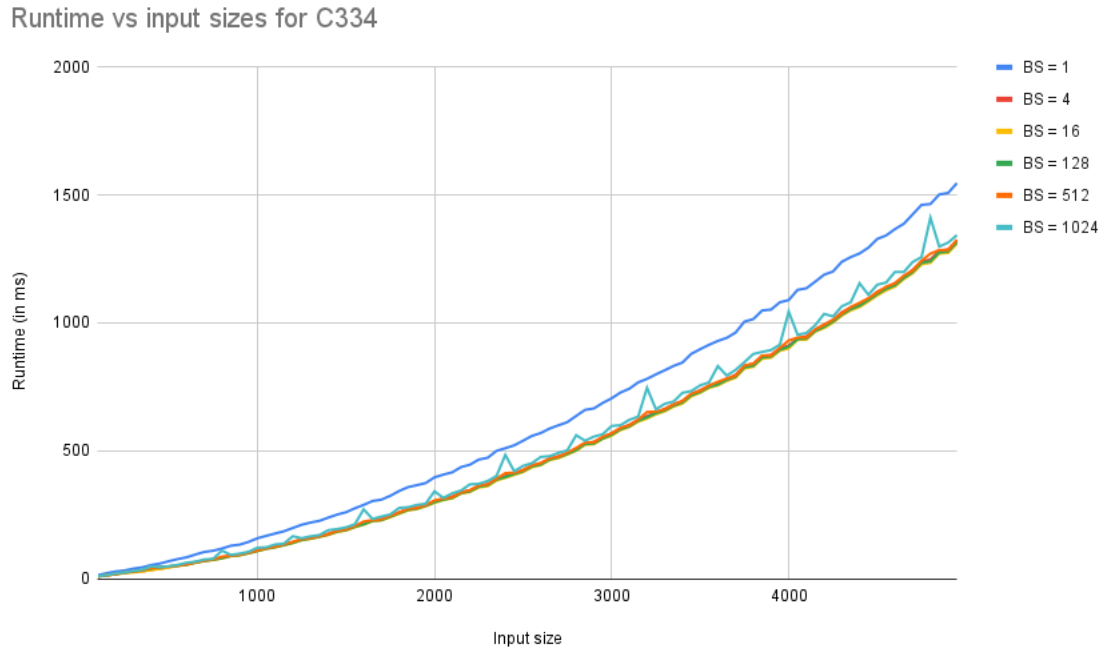

FIGURE 7. Runtime vs input for for ShortLex representative in C_{334}.

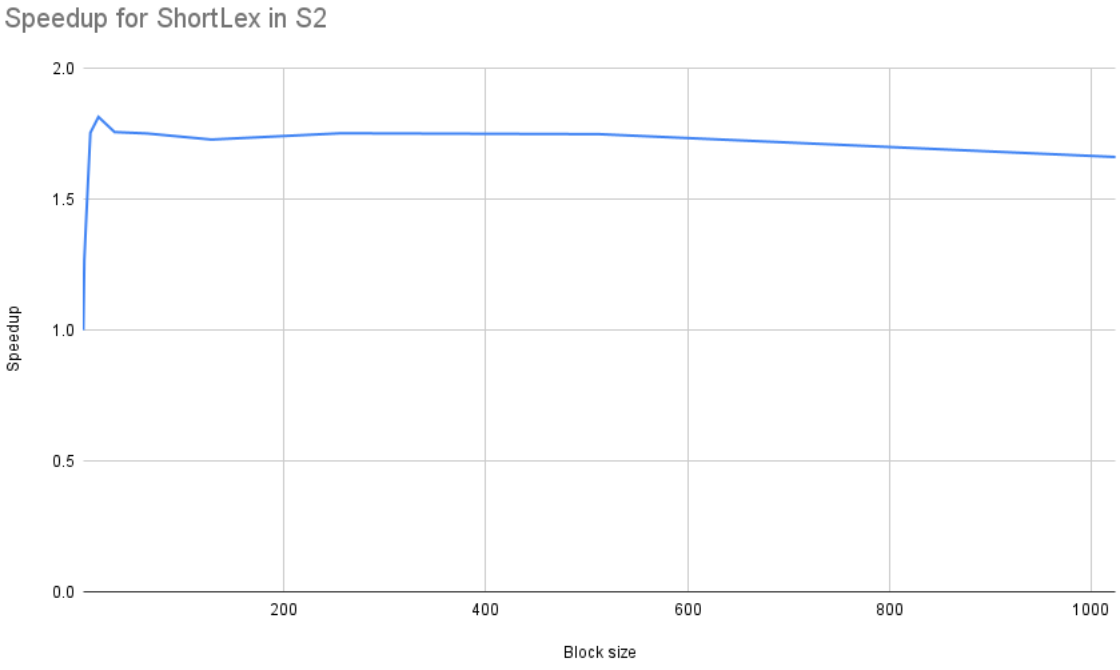

FIGURE 8. Speedup for ShortLex in S_2.

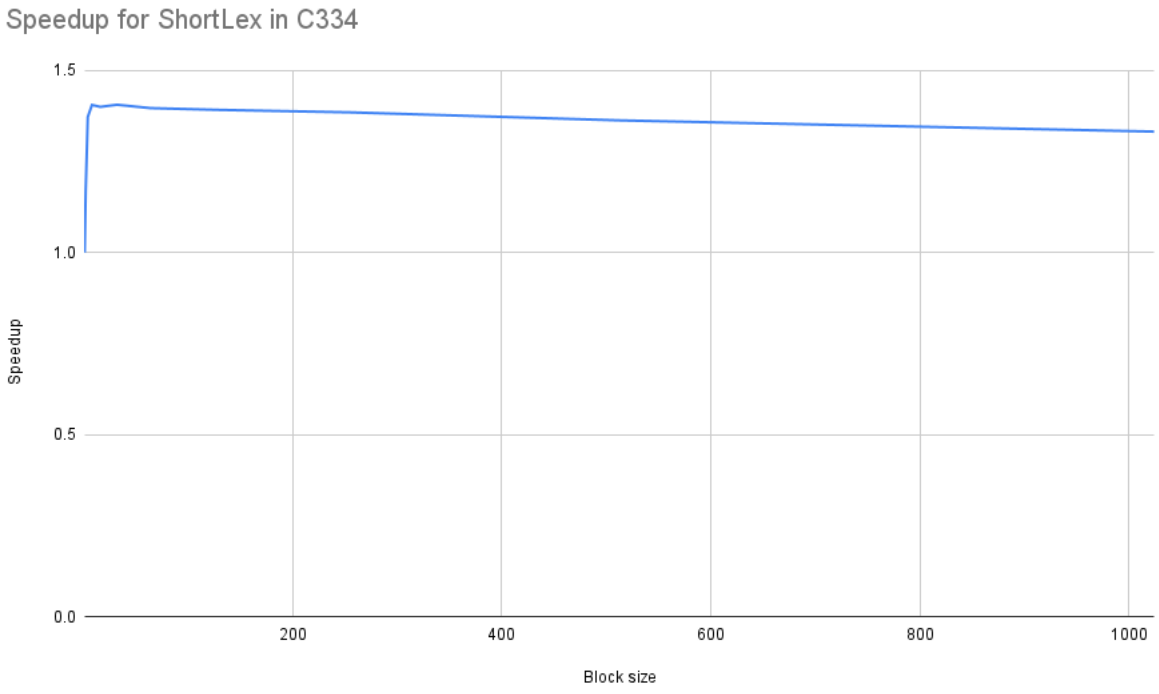

FIGURE 9. Speedup for ShortLex in C_{334}.