Parallel All Nearest Smaller Values and Range Minimum Query on the GPU

A step towards parallel lossless compression on the GPU

Jonas Ryssel (s184009)

25th June 2025

Technical University of Denmark

Supervisors: Inge Li Gørtz
Philip Bille

Contents

1 Abstract

We study two algorithms relevant for designing a parallelised Lempel-Ziv-77 compression scheme. First, we optimise the All Nearest Smaller Values algorithm from (Berkman et al., 1993), and then we design a variant of it that is suitable for running on a GPU while performing $O(n)$ work with $O(\log^2(n))$ span on $\frac{n}{\lg(n)}$ threads on a CREW PRAM. When implemented, our GPU version achieves a 4-5 times speed-up over our CPU version.

We then design a GPU version of the precomputation of Lowest Common Ancestor queries described in (Schieber & Vishkin, 1988) in order to answer Range Minimum Query queries in constant time. It performs $O(n)$ work and has $O(\sqrt{n \log(n)})$ span on $\sqrt{\frac{n}{\lg(n)}} - 1$ threads on a CREW PRAM.

2 Introduction

Lossless compression algorithms are algorithms designed to take a large input and compress (shrink) it in a way where the original input can be restored from just the compressed data. This is in contrary to lossy compression algorithms, which allows discarding less significant data in order to compress the data more, but in turn only manages to restore an approximation on the original input.

A common property of lossless compression algorithms is that it takes much longer to compress data than it takes to decompress it. This is because, roughly speaking, when we compress data we need to search for patterns in the data (e.g. a sentence in a book repeating multiple times) but when we decompress data we are given a list of patterns that we just need to execute in reverse (e.g. replace each instance of a token with the sentence that it represents), which is much faster.

We therefore started looking for ways to speed up the compression process by trying to parallelise the compression part of a compression algorithm. An easy way to parallelise any compression algorithm is to split the input data into parts, and compress each of them individually. This, however, may result in worse compression, since we no longer have a way to share information across the parts of the input data. It is also technically a different compression algorithm, since the output of running the algorithm on the entire input data, and on parts of the input data are different. We therefore specifically wanted to find an parallelisation of a compression algorithm, which returns the same output as the original algorithm.

In (Shun & Zhao, 2013b) they found a parallelisation of the Lempel-Ziv-77 (LZ77) algorithm, which achieved high levels of speed-up when run on a multi-core Central Processing Unit (CPU). One of the steps in that algorithm is to run a parallel all nearest smaller values algorithm (ANSV) along with answering range minimum queries (RMQ). In this thesis, we focus on improving these two steps.

For the ANSV problem, they cite (Berkman et al., 1993) which presents an algorithm that can parallelise on up to $\frac{n}{\lg(n)}$ threads. A CPU, however, only has relatively few cores, meaning that it can only execute a few threads at a time. In contrary, a Graphics Processing Unit (GPU) is specifically designed to run highly parallelised calculations on many cores, but in return it suffers from limitations that means that it cannot execute regular algorithms. We therefore focus on converting the algorithm from (Berkman et al., 1993) into one that can run on a GPU.

(Berkman et al., 1993) also mentions an algorithm for answering RMQ queries in constant time, using data precomputed from the output of the ANSV algorithm. We therefore decided to also convert this algorithm to a version capable of running on a GPU.

Part I
All Nearest Smaller Values

3 Defining the problem and relevant variables

Before we start explaining the algorithm, we first need to define the problem.

Definition 3.1. Given an array of integers A, find the following for all elements in A:

- For the element at index k, find the element at index i which satisfies, $i < k$, $A[i] < A[k]$, $\forall \ell \in \mathbb{N}, i < \ell < k : A[\ell] \geq A[k]$. This is referred to as the "left nearest smaller value", or just the "left match".
- For the element at index k, find the element at index j which satisfies, $j > k$, $A[j] < A[k]$, $\forall \ell \in \mathbb{N}, k < \ell < j : A[\ell] \geq A[k]$. This is referred to as the "right nearest smaller value", or just the "right match".

It is possible for an element to be missing its left-, right-, or both matches

The other thing we need to explain is some variable names and terms that we will be using in order to make the explanations more readable. These variable names and terms are chosen to be consistent with the ones used in (Berkman et al., 1993).

- The index of the "main" element currently focused on by a thread is k.
- The index of an element left of k is i, with "left of" meaning that $i < k$.
- The index of an element right of k is j, with "right of" meaning that $j > k$.
- During the algorithm we split A into non-overlapping parts called subsets.
- An element's index in A is its global index.
- An element's index in its subset is its local index.
- The global index of the minimum element in subset k is b_k.
- The global index of the left match of b_k is $l(b_k)$.
- The global index of the right match of b_k is $r(b_k)$.
- A "nonlarger value" is defined similarly to a "smaller value" but where we look for an element of equal or smaller value. The definition of the left nonlarger value would e.g. be $i < k$, $A[i] \leq A[k]$, $\forall \ell \in \mathbb{N}, i < \ell < k : A[\ell] > A[k]$.
- The global index of the left- and right nearest nonlarger value of b_k are $nl(b_k)$ and $nr(b_k)$.
- The index of the subset containing element $l(b_k)$ is $gl(b_k)$. We sometimes replace this with i, allowing us to write $r(b_i)$ instead of $r(b_{gl(b_k)})$. Similarly the subset containing $r(b_k)$ is $gr(b_k)$ and we sometimes write $l(b_j)$ instead of $l(b_{gr(b_k)})$.
- In the same way, the subsets containing $nl(b_k)$ and $nr(b_k)$ are $gnl(b_k)$ and $gnr(b_k)$.

3.1 Computation model and analysis terms

All the algorithms described in this thesis (both ANSV and RMQ algorithms) share the property that they split up their calculations into non-overlapping parts, before the assign them to the threads for parallel execution. This means that while two processors might read overlapping input and calculate the same variables local to themselves, they each calculate and write non-overlapping parts of the output array for any given subroutine.

This means that the algorithms are compliant with the Concurrent Read Exclusive Write PRAM (CREW PRAM) model.

For the analysis of the algorithms, we use the terms work and span. Work is the number of calculations performed in total across all threads. Span is the maximum number of calculations performed by any thread.

To compare this to the usual runtime analysis of a sequential algorithm, the span is the actual runtime of the parallel algorithm, since the algorithm takes only as long as the slowest thread.

The work is how long it would take to run the parallel algorithm if we ran it sequentially, with one thread performing all the work.

4 The ANSV algorithm from (Berkman et al., 1993)

In short, the algorithm described in (Berkman et al., 1993) works by splitting A into subsets, finding all matches within the subsets, and then finding all matches across subsets. This way, it can achieve parallelisation over $\frac{n}{\lg(n)}$ threads by splitting A into $\frac{n}{\lg(n)}$ subsets of size $\lg(n)$ each.

Some of the smaller steps in the algorithm are repeated multiple times, why they are split into subroutines. We will explain the subroutines first, and then the full algorithm.

The standard version of the algorithm assumes that all values in the input array A are distinct.

4.1 Basic search procedure

The first subroutine is the *basic search procedure*. It works by first constructing a binary tree with the leaf nodes being the elements of A and the other nodes having the value of the minimum of its descendants. Henceforth, we refer to such a tree as a min-tree. Once the min-tree is constructed, we can find the nearest smaller value to the left of a leaf node with value a by walking up the tree until the left sibling is less than a. We then go to the left sibling and walk down the tree going to the right child if it is less than a, and otherwise going to the left child.

To instead find the right nearest smaller value, swap all references of left and right. You can also find the nearest nonlarger values by replacing all "less than" checks with "less than or equal".

Constructing the min-tree takes $O(n)$ work and $O(\log(n))$ span, and running the basic search procedure is single threaded and runs in $O(\log(n))$ time.

4.2 The merging procedure

The second subroutine is the *merging procedure*. It takes two arrays A_1 and A_2 as input, and finds matches for elements across the arrays.

To make the concept of matches make sense, we assume that A_1 and A_2 are non-overlapping sub-arrays originating from the same array, and that A_1 is located left of A_2. This means that we are looking for right matches for elements in A_1 and left matches for elements in A_2. We also assume that all elements in the input arrays are distinct.

The merging procedure starts with calculating the suffix minima of A_1 and naming it C_1, then calculating the prefix minima of A_2 and naming it C_2. It then merges C_1 and C_2 into a monotonic increasing list C.

We now notice that for any element e in C, if we assume that e originates from C_1, we can calculate the number of elements in C_2 that are smaller than e by subtracting e's index in C_1 from its index in C. It also works the other way if e instead originates from C_2. This means that we can calculate the matches across the arrays using the following formulas:

- Let ℓ be the length of C_2.
- Let r refer to an element's own index in C_1 or C_2.
- Let m refer to an element's own index in C.
- If $r = m$ (i.e. the element and all previous elements in C are from the same array), there is no match.
- If the element originates from C_1, its right match is $\ell - (m - r)$.
- If the element originates from C_2, its left match is $m - (\ell - r)$.

This obviously runs in linear time.

The reason why we assume that the elements are distinct, can be seen if we place an element of the same value in each array. If the input is $A_1 = [1,3]$ and $A_2 = [3,2]$, then the merged list will be either $C = [1,2,3,3]$ or $C = [1,2,3,3]$, with green denoting A_1 and red denoting A_2. In either case, one of the elements of value 3 will get a match of value 3, which is not a smaller value.

4.3 The full ANSV algorithm

With the two subroutines defined, we can now explain the full ANSV algorithm. We start by creating the min-tree, and splitting the array into $\frac{n}{\lg(n)}$ subsets of length $\lg(n)$ to which we assign one thread each. The rest of the algorithm is explained as a sequential algorithm from the point of view of thread k assigned to subset k. The parallelism then follows naturally, since we have $\frac{n}{\lg(n)}$ threads.

Whenever one thread reads a value written by another thread, we implicitly synchronize the threads, such that they all catch up to each other and the value is written before it is read. In practice it is enough to synchronize the threads twice. We also implicitly convert all results local to a subset (such as the index of the minimum element) to global indices before we save them to the arrays shared between threads. Converting a local index to a global index and back to a local index can be done by adding $k\lg(n)$ and subtracting $\left\lfloor \frac{index}{\lg(n)} \right\rfloor \lg(n)$.

Subset specific calculations

The first thing a thread does is to perform calculations local to its subset. Specifically:

- Solving the ANSV problem locally using the standard sequential stack-based algorithm.
- Finding the index of the minimum element, b_k.
- Calculating the prefix minima.
- Calculating the suffix minima.

Global calculations
The second thing the thread does is to precompute values necessary for solving the ANSV problem across subsets. Specifically:

- Finding the nearest smaller values of b_k, namely $l(b_k)$ and $r(b_k)$ using the basic search procedure.
- Calculating the subsets that these elements belong to. That is $gl(b_k) = \left\lfloor \frac{l(b_k)}{\lg(n)} \right\rfloor$ and $gr(b_k) = \left\lfloor \frac{r(b_k)}{\lg(n)} \right\rfloor$.

With everything ready, the thread can now use the merging procedure to match elements across subsets. The thread is responsible for checking four conditions, and depending on which ones are true, running the merging procedure on the prefix- and suffix minima of various subarrays of the full array. We always use the suffix minima of the leftmost subarray and the prefix minima of the rightmost subarray. The four cases are:

1. If $gl(b_k) = k-1$, we run the merging procedure on subarrays $l(b_k) \dots k\lg(n)-1$ and $k\lg(n) \dots b_k$.
2. If $gr(b_k) = k+1$, we run the merging procedure on subarrays $b_k \dots (k+1)\lg(n)-1$ and $(k+1)\lg(n) \dots r(b_k)$.
3. If $l(b_k)$ and $r(b_k)$ were both found, and $l(b_k) < r(b_k)$, we run the merging procedure on subarrays $l(b_j) \dots l(b_k)$ and $r(b_k) \dots b_j$.
4. If $l(b_k)$ and $r(b_k)$ were both found, and $l(b_k) > r(b_k)$, we run the merging procedure on subarrays $b_i \dots l(b_k)$ and $r(b_k) \dots r(b_i)$.

We note that since each of the subarrays is contained within a single subset, and since the start/end index of each subarray is either the start/end of a subset (depending on if we use the prefix- or suffix minima) or the nearest smaller value of an element outside the subarray, the prefix-/suffix minima of the subarray will always be the same as the same indexes of the prefix-/suffix minima of the entire subset. We can therefore use the prefix-/suffix minima that we calculated beforehand.

The output of the merging procedure is the matches of the elements that are the same in the subset and in the prefix-/suffix minima of the subset. For all other elements, the fact that they have a different value in the prefix-/suffix minima means that its nearest smaller value is within its own subset. We can therefore find the entire ANSV solution by taking all the local ANSV solutions, and for all the elements that do not have a local match, use the match found via the merging procedures (which we convert to global indices by adding the start index of the respective subarray). Any remaining elements that we still have not found a match for simply do not have matches.

Looking at the individual parts of the algorithm, we see that:

- Constructing the min-tree performs $O(n)$ work and has $O(\log(n))$ span on $\frac{n}{\lg(n)}$ threads.
- Finding $l(b_k)$ and $r(b_k)$ is single threaded and takes $O(\log(n))$ time.
- Everything else is single threaded and runs in $O(m)$ time, but is only executed on inputs of length $m = O(\log(n))$, hence running in $O(\log(n))$ time.

In total, since each thread performs $O(\log(n))$ work and we spawn $\frac{n}{\lg(n)}$ threads, the algorithm performs $O(n)$ work with $O(\log(n))$ span on $\frac{n}{\lg(n)}$ threads.

5 Generalised CPU version

So far we have assumed that all elements in the input array are distinct. The reasons for this are that otherwise:

1. The merging procedure would return an incorrect output, as seen in the example shown in section 4.2.
2. The method for selecting which subarrays to run the merging procedure on would no longer ensure that all elements find their matches.
3. In the original article, it is shown that only one thread will run the merging procedure on each pair of subarrays.[1] While this does not affect the correctness of the algorithm, it would turn the computation model from CREW to CRCW due to multiple threads potentially writing the same value.
4. We assume that there is an unique minimum element in each subset.

The method to solve this problem suggested in (Berkman et al., 1993) is to enumerate all values in the array, turning them into tuples, with the index being a tie breaker to decide which of two identical elements is "smaller". When we enumerate the values with increasing indices, all right matches will be correct, but the left matches risk pointing to elements with the same value. We therefore also reverse the enumeration, since with decreasing indices, it is now then left matches that are correct. This means that by modifying the input and running the algorithm twice, we can handle duplicate values.

We will present a modified version of the algorithm which does not require modification of the input and only needs to be run once, yet still handles duplicate values.

In order to fix finding matches across subsets, we will present a modified merging procedure and a modified set of rules for which subarrays to run the merging procedure on. The alternative merging procedure fixes the correctness of the output when the input includes duplicates, while the rules for which subarrays to run the merging procedure on ensures that we correctly run the merging procedure on all relevant subarrays while staying within the CREW PRAM computation model.

Finally, in order to handle multiple minima within a single subset, we will present a trick to have those multiple minima act as a single minimum. Since we explain this last, for the earlier parts of the algorithm, we assume that each subset has a single minimum element, b_k.

5.1 The modified merging procedure

The input to the modified merging procedure is two arrays A_1 and A_2.

We start by filtering out all elements that already know their match (right match for elements in A_1 and left match for elements in A_2). This can be done by either passing the information along as extra input, or by calculating the prefix-/suffix minima and only keeping the elements where their prefix-/suffix minima is equal to itself. The remaining elements are stored in arrays C_1 and C_2 along with their indices from A_1 and A_2. We then merge C_1 and C_2 into a monotonic increasing list C.

We can then find the matches by iterating over C and using the following logic:

- Let (val,$index$) refer to an element's value and index from A_1/A_2.
- Let a be the element we are currently trying to find a match for.

[1]Berkman et al., 1993, Lemma 3.1, page 351.

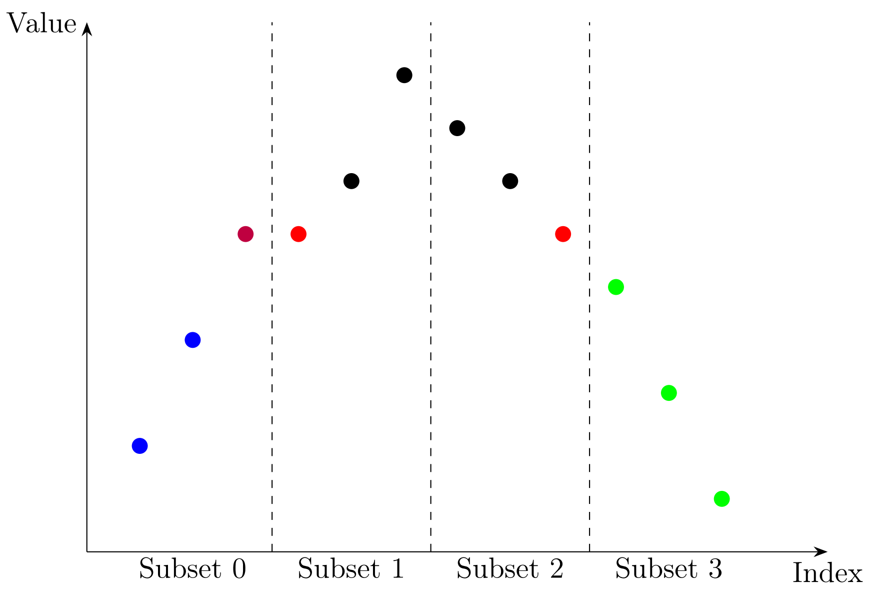

Figure 1: A visualization of an array where the original ANSV algorithm would fail to find all the correct matches. The vertical axis represents the value of the elements, the horizontal line represents the index, and the vertical lines represent the boundaries between subsets. An example of an array that would fit this illustration is [1,3,5,5,6,8,7,6,5,4,2,0].

- Let b and c refer to the last two elements with distinct values from the other array. I.e. if a is from C_1, and we have previously seen (1,1), (2,3), and (2,4) from C_2, then $b = (1,1)$ and $c = (2,4)$.
- If the value of a is greater than the value of c, the match for a is the index of c. Otherwise the match for a is the index of b.
- If we cannot pick either b or c to be the match of a due to not yet having iterated over elements from the other array, then a does not have a match to be found in the input to the merging procedure.

5.2 Selecting subarrays to merge

To know what to modify in the logic of selecting which subarrays to merge, we start by looking at an example of what situations the original algorithm fails to calculate the correct output.

Looking at fig. 1 we can see three issues:

(a) Since the minimum in subset 1 and 2 are the same value (the red dots), both thread 1 and 2 run the merging procedure on the subarrays coloured blue and green.

(b) The merging procedure is run on the subarrays coloured blue and green, but to get the correct output, the purple element should have been part of the blue subarray.

(c) No thread runs the merging procedure on the elements of subset 1 and 2, despite the fact that the merging procedure should be run on the red and black elements.

Informally, the three issues can be solved in the following way:

(a) In case where the minima of multiple neighbouring subsets are the same, only the leftmost subset runs the merging procedure.

(b) When using the basic search procedure to find the start and end indices for the subarrays that should be merged, instead of looking for the nearest smaller value, we look for the nearest nonlarger value.

(c) When checking if a subset should be merged with its neighbouring subset to the left, instead of checking if the nearest smaller value is in the subset, check if the nearest nonlarger value is in it. Note that when comparing to the neighbouring subset to the right, we still check for the nearest smaller value in order to avoid duplicated work.
In case we merge subarrays from two neighbouring subsets with the same minimum value, we no longer find the matches of the minimum elements in the direction of the other subset. To remedy this, we use the basic search procedure.

Formally, the new rules for deciding which subarrays to run the merging procedure on look as follows:

- If $gnl(b_k) = k - 1$ we run the merging procedure on subarrays $nl(b_k) \dots k\log(n) - 1$ and $k\log(n) \dots b_k$.
- If $gr(b_k) = k + 1$, we run the merging procedure on subarrays $b_k \dots (k+1)\log(n) - 1$ and $(k+1)\log(n) \dots r(b_k)$.
- If $A[b_{k-1}] = A[b_k]$, find the right match of b_{k-1} using the min-tree.
- If $A[b_{k+1}] = A[b_k]$, find the left match of b_{k+1} using the min-tree.
- If $l(b_k)$ and $r(b_k)$ both exist, and both $gl(b_k) = gnl(b_k)$ and $gnr(b_i) = gr(b_k)$, then we:
    1. Find the nearest nonlarger element to the right of element b_k in subset j, and denote it $nr_j(b_k)$. This element is found using the min-tree as usual, but by starting the search in element $j\log n - 1$ while ignoring the value of the element itself and instead searching for the value of element b_k.
    2. Run the merging procedure on subarrays $b_i \dots nl(b_k)$ and $nr_j(b_k) \dots nr(b_i)$.
    3. If $A[b_i] = A[nr(b_i)]$, find the right match of b_i, $r(b_i)$, using the min-tree.
- If $l(b_k)$ and $r(b_k)$ were both found, $gr(b_k) = gnr(b_k)$, $gl(b_j) = gl(b_k)$, then we:
    1. Find the nearest nonlarger element to the left of element b_k in subset i, and denote it $nl_i(b_k)$. This element is found using the min-tree as usual, but by starting the search in element $(i+1)\log n$ while ignoring the value of the element itself and instead searching for the value of element b_k.
    2. Run the merging procedure on subarrays $l(b_j) \dots nl(b_k)$ and $nr_j(b_k) \dots b_j$.

We would like to point out that while the two rules for when $l(b_k)$ and $r(b_k)$ are both found look very similar, their differences are very important. The difference in their usage of smaller values and nonlarger values is what ensures that two threads do not perform overlapping work. I.e. it is what guarantees that the algorithm stays in the CREW PRAM computation model.

5.3 Multiple minima within a subset

So far we have assumed that each subset contains only a single minimum element, b_k, to keep the description simple. We will now change the meaning of b_k depending on the context to handle multiple minima elements while preserving the correctness of the rest of the algorithm.

We observe the following properties of the algorithm with regards to having multiple minima elements within a single subset:

- If two or more minima elements are present in a subset, all elements in-between these elements (except minima elements) will have their ANSV found by the sequential algorithm.
  The logic behind this property is that since these elements have a minimum element both to the left and to the right of them, they have a smaller element on both sides of them, meaning that the ANSV algorithm will find a match to both sides.

- When part of a subset is selected as the left input array to the merging procedure, only the elements to the right of the rightmost minima element are relevant to the merging.
  For this, we point out that since all elements to the left of the rightmost minima element already have right matches within the subset, these elements cannot find matches in a neighbouring subset. Likewise, the elements in the neighbouring subset cannot find their match to be the left of the rightmost minima element. The only exceptions are the other minima elements which matches must exist outside the subset, but those are handled by another property.

- The same is true, where the right input array of the merging procedure only needs elements to the left of the leftmost minima element.

- The left and right matches for all minima elements in a subset will be the same.
  The logic behind this property, is that since there are no smaller values in-between the minima elements, they closest smaller element outside the subset will be the closest smaller element to all the minima elements.

Using these properties, we declare the following rules about which minimum element is represented by b_k:

- When the right input array for the merging procedure is selected from the subset, b_k represents the leftmost minimum element.

- When the left input array for the merging procedure is selected from the subset, b_k represents the rightmost minimum element.

- When we use the basic search procedure to find $l(b_k)$ or $nl(b_k)$, b_k represents the leftmost minimum element.

- When we use the basic search procedure to find $r(b_k)$ or $nr(b_k)$, b_k represents the rightmost minimum element.

This way we have effectively contracted all the minima elements and all the elements in-between into a single point, while finding matches across subsets. After we are done running the merging procedure on all relevant subarrays, we execute the following logic on all subsets to finish the algorithm: Take the right match of the rightmost minimum element and the left match of the leftmost minimum element and copy those to all the other minima elements. This way, all minima within a subset, find the same left and right matches, just as they should.

5.4 Algorithmic properties

While the modified algorithm is significantly more complicated due to it needing to handle many more special cases, it does not affect the algorithmic properties of the algorithm.

The merging procedure still runs in linear time of its input size. The merging procedure is still only run a constant number of times per thread. The basic search procedure is also still only run a constant number of times per tread. And finally, copying the ANSV result amongst the multiple minima elements within a subset, also runs in linear time per thread.

This means that the work is still $O(n)$, the span is still $O(\log(n))$ and the number of threads used is also still $\frac{n}{\lg(n)}$.

6 The BSZ algorithm

Later, when we compare the performance of our ANSV algorithm to previous implementations, we use the implementation of the ANSV algorithm made for and used in (Shun & Zhao, 2013b).[2] This implementation uses a heuristic to simplify the algorithm so we will explain said heuristic. In (Sitchinava & Svenning, 2024) they call this the BSZ algorithm, which is the name that we will adopt.

The algorithm starts by splitting A into s subsets and finding the ANSV locally within each subset using a sequential algorithm. It then constructs the min-tree.

The third step is to find the right matches using the following procedure:

1. Take the rightmost unmatched element a in a subset and find its right match using the min-tree (as in the basic search procedure).

2. Take the next unmatched element b to the left of a, and find its right match using the min-tree. This time, however, we do not start the search in b, but instead we start at $r(a)$.

3. Repeat step 2 for all unmatched elements in the subset.

This step is shown in fig. 2.

The fourth and final step is to find all the left matches, using the same method as for the right matches, but where we search left instead of right.

The reason why the heuristic works can be seen if we look at the example shown in fig. 2. We know that $A[b] \leq A[a]$ since otherwise b would find a to be its right match within the subset. We also know that $\forall j \in \mathbb{N}, a < j < r(a) : A[j] \geq A[a]$ since otherwise a would have found a right match before $r(a)$. By transitivity, we now know that $\forall j \in \mathbb{N}, a < j < r(a) : A[j] \geq A[b]$, which means that $r(b) \geq r(a)$, so there is no point in searching the part of the array left of $r(a)$ when we look for the right match of b.

In (Sitchinava & Svenning, 2024) they show that for a $1 \leq s \leq n$, this algorithm achieves $O\left(n\left(1 + \frac{\log(n)}{k}\right)\right)$ work and $O\left(k\left(1 + \log\left(\frac{n}{k}\right)\right)\right)$ span.[3] By setting $k = \Theta(\log(n))$ they claim that it achieves $O(1)$ work and $O\left(\log(n)\log\left(\frac{n}{k}\right)\right)$ span.

[2]Shun and Zhao, 2013a.

[3]Sitchinava and Svenning, 2024, p. 261.

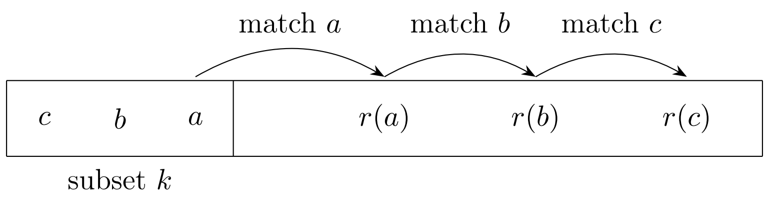

Figure 2: Illustration of the heuristic used in (Shun & Zhao, 2013b). In subset k, the elements a, b, and c have yet to find their right matches. The arrows beneath "match a", "match b", and "match c" show the parts of the array that are searched using the min-tree when looking for the right matches of a, b, and c respectively.

7 GPU version

In this section, we will present a version of the ANSV algorithm tailored for running on a GPU. The modified merging procedure shown in section 5.1 did not turn out to be suitable for a GPU, so we based it on the version of the ANSV algorithm explain in (Berkman et al., 1993).

7.1 How a GPU works / GPU Terminology

Before we explain our GPU algorithm, we will roughly explain how GPU hardware is designed and some of the features it has. This will not be a comprehensive explanation, but the aim is to simplify and explain the parts that are necessary in order to understand the algorithms and the performance characteristics we will touch on in the discussion. It shall also be noted that much of the GPU terminology is dependent on which company is designing the GPU, but the actual hardware is roughly the same just with different marketing. In our case, we will use the terminology used by AMD.

7.2 Workgroups and kernels

On a GPU we have *workgroups* which are sets of 32 threads grouped together (the exact number varies depending on the hardware). All the threads in a workgroup sharing the same set of instructions and are started and stopped together. When running code, each workgroup knows what its *workgroup index* is, and each thread in each workgroup knows what its *thread index* is. The threads can therefore perform independent work by calculating which indices of the input and output arrays that it should read from and write to.

This is different from how a CPU works, where each thread has its own set of instructions. This means that while it is possible to run a CPU algorithm on GPU hardware by starting the algorithm with "if thread index is 0, execute this", this would result in 31 of the 32 threads doing nothing while waiting for the one thread to finish. Instead GPU algorithms should be designed in such a way that all 32 threads perform approximately the same amount of work and finish at the same time.

A piece of code designed for running on a workgroup is called a *kernel*. When we *dispatch* (run) a kernel, we run many workgroups at a time. The actual hardware that executes the threads in a workgroup is called a *SIMD*. A GPU is designed to have as many SIMDs as possible, typically hundreds.

7.2.1 Synchronization

One important consideration when designing GPU algorithms is *synchronization*. When executing a kernel across multiple workgroups, we do not have any guarantees on which order the operations are executed in. This means that if two threads need to collaborate, neither of them can know when the result from the other thread has been written to an array.

Consider for example if we want to find the minimum of an array. Each workgroup is assigned part of the array, and each thread in said workgroup is then assigned part of that part. After all the threads have found the local minima of their part, we still need to somehow find the minimum across all the parts. Naively, we could just let one thread read all the local minima and find the minima amongst them, but even this does not work, since we have no way to know when each thread has written their local minima have been written to a shared array.

The solution to solve this problem is to synchronize the threads and workgroups by utilizing *barriers*. Depending on the hardware configuration there exist many types of barriers, but the two most common ones are the workgroup-wide barrier, which synchronizes the threads within a single workgroup, and the implicit synchronization that happens after a kernel is done executing.

The workgroup-wide barrier makes it easy and fast to share information between threads in a workgroup. The synchronization between two kernels being dispatched is useful for implementing more complex algorithms, where we need synchronization between all threads.

Whenever we explicitly refer to synchronization in the algorithms, we mean workgroup-wide synchronization, as the kernel synchronization happens implicitly.

7.2.2 Memory and cache layers

While it varies slightly over time and between GPU manufactureres, the general design of a GPU's memory hierarchy is as follows:

- Video RAM (VRAM) is the GPU's main memory.
- The level 2 cache is the next cache layer with slightly faster cache. Not all SIMDs share the same level 2 cache.
- The Local Data Share (LDS) is the smallest and fastest cache on a GPU. Each LDS cache is shared between four SIMDs.

While a SIMD unit can only execute one workgroup at a time, it is designed to store up to 16 workgroups in its cache. Then, whenever a workgroup needs to access some data not in the LDS cache, the SIMD can quickly switch to working on another of the 16 workgroups, while it waits for the data to be retrieved from higher up in the memory hierarchy.

This is one of the tricks a GPU has to speed up its execution, but in return it limits us when we design our algorithm since we need to be very aware of the cache usage of our algorithm. As if a workgroup requires too much data to be cached, then the SIMD unit first stops preparing the other 15 workgroups, and if it is even worse, then the other three SIMD units that share the same LDS cache stop preparing their workgroups as well.

7.3 Assumptions and tricks

We need to point out a two assumptions that we make. They are strictly speaking irrelevant to the algorithm itself, and lean more towards implementing the algorithm in practice, but they straighten a couple rough edges in the explanations.

- We assume that unless otherwise mentioned, all arrays use unsigned 32 bit integers as their data type.

  The reason why we assume a data type is that we need to know the minimum value and the maximum value that an input element can have.

- Keeping track of which elements have found matches, and which have not can be tricky in GPU code. To reduce the need for a separate array to store this data (which simplifies our explanations), we define an index being the maximum possible value of an unsigned 32 bit integer to mean that we have not found a match for said element.

7.4 Common algorithm patterns on GPUs

In order to facilitate simpler descriptions of the GPU kernels, we will start by describing some smaller algorithms and techniques that are useful on a GPU. These can then be used as parts of GPU kernels without further explanation.

7.4.1 Tree reduction

Often, when we e.g. want to find the minimum element in an array, we split the array into parts and assign each part to a thread which performs the comparisons. After this, we now have a minimum for each thread, but we still need to find the minimum across thread. To accomplish this, it is common to reduce the thread minima in a binary tree structure. Let the leaves be the threads minima, and the root being the global minima. For each layer in the tree, we take every node and let the left child (thread) find the minimum of both the children, and write the result to the node. Once all nodes in a layer are reduced, we start with the next layer.

The tree structure and thread assignment is described formally in listing 1 and visualised in fig. 3.

This method is typically used for the 32 threads within a workgroup, as the workgroup synchronization operation is fast, so it does not matter if we use it five times. If the method is used across workgroups, then the upper bound on the number of leafs in the tree is much higher and synchronization only happens when a kernel finishes running, so we need to dispatch multiple kernels which is very expensive. In this case, we let each node in the tree have the same number of children as in the original array parts, to reduce the number of kernels that needs to be dispatched.

If the number of threads is constant, calculating the thread minima results in $O(n)$ work and span, while the tree has a constant number of nodes and therefore does not affect neither work nor span.

If instead the size of the part of the array assigned to each thread is constant, calculating the thread minima results in $O(n)$ work and $O(1)$ span, while the tree now has a linear number of nodes, so reducing the tree results in $O(n)$ work and $O(\log(n))$ span.

We can therefore conclude that performing tree reduction requires $O(n)$ work and $O(n)$ or $O(\log(n))$ span depending on if the number of threads or the number of elements per thread is set to be constant.

```
t: The index of the current thread
M: The array containing the thread minima

For i=0..$\lceil \lg(T) \rceil$-1:
    d = $2^i$
    If $t \equiv 0 \mod d$ and $t + d < T$:
        M[t] = min(M[t], M[t+d])
    Synchronize
```

Listing 1: Tree reduction

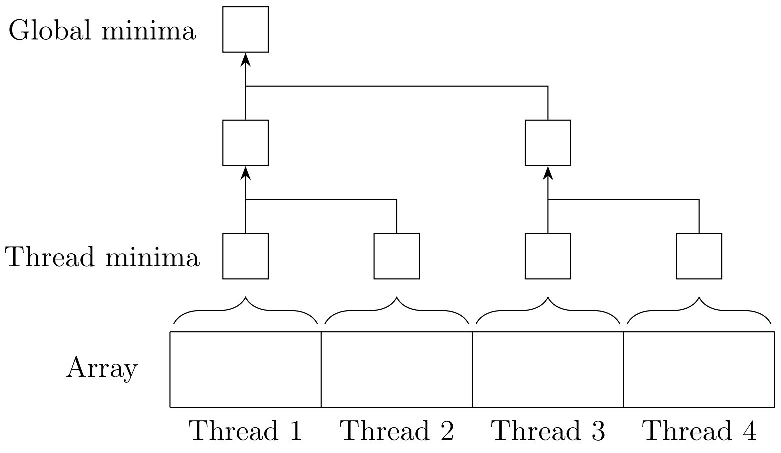

Figure 3: The bottom array is split into four parts and each part is assigned to a thread. Each thread then finds the minima in its part, after which the minima across threads is found.

7.4.2 Single workgroup minimum

As explained section 7.4.1, a single workgroup can find the minimum of an array by splitting the array into 32 parts, having each thread find the minimum of one part, and then performing tree reduction to find the global minimum.

As we have a constant number of threads, doing this performs $O(n)$ work and has $O(n)$ span.

7.4.3 Single workgroup prefix- and suffix minima

To find the prefix minima of an array, start by splitting the array into 32 parts and have each thread find the minimum of their part. We then synchronize to allow all threads to know all the thread minima. Each thread now finds the minima of all previous threads, and finds the prefix minima of its own part using a sequential algorithm, but with the initial prefix minima being the minima of the previous threads. The algorithm is described formally in listing 2.

Since all the algorithm does is that each thread iterates over its part of the array twice, and over the constant length thread-minima array once, the work and span are $O(n)$.

The suffix minima is found similarly.

```
t: The index of the current thread
e: The number of elements per thread
A: The input array
M: The array containing the thread minima
P: The array containing the prefix minima

```

```
M[t] = min(A[t*e..(t+1)*e-1])
Synchronize
p = min(M[0..t-1])
For i=0..e-1:
    p = min(p, A[t*e+i])
    P[t*e+i] = p
```

Listing 2: Prefix minima

7.4.4 Bitonic merge

To emulate the merging procedure on a GPU, we need to be able to merge a monotonic increasing array and a monotonic decreasing array into a single monotonic increasing array. We let the input be a single array that is the concatenation of the monotonic increasing array and the monotonic decreasing array (henceforth named *array*), along with the index of when the array switches from monotonic increasing to monotonic decreasing (the peak). A sequence that is first monotonic increasing and then monotonic decreasing is called a bitonic sequence, hence why we name this a bitonic merge.

We add the extra assumption that no value is shared between the monotonic increasing and monotonic decreasing parts of the input array. That is, we assume that the input is of the form that we would get by having an array with no duplicate values, and then finding the suffix minima on the half left of the peak, and prefix minima on the half right of the peak. This assumption is necessary in order to for there to only exist one valid merged list, which we need in order to avoid the same issues we discussed regarding the generalized CPU version of the ANSV algorithm.

In a sequential algorithm, we would start at the peak and iterate to the left and right picking whatever value is the largest and writing that to the output array starting from the end (such that the start of the output array contains the smallest values). To parallelise this on a GPU we want to predict which indices the sequential algorithm would compare when deciding what value to write to a specific index in the merged list. Formally, we want to find the indices i_1 and i_2 of the two elements that are compared when writing index j of the merged array.

To locate these indices, each thread runs two concurrent binary searches. One locates i_1 in the monotonic increasing part, i.e. $-1 \ldots peak - 1$, and the other one locates i_2 in the monotonic decreasing part, i.e. $peak \ldots |array|$ (with i_1 and i_2 being stored as signed integers). It is on purpose that we allow indices outside the array, since we need these indices to be hardcoded to certain values based on which binary search is reading them:

- If $i_1 = peak$ then $array[i_1 + 1] =$ the maximum value storable by an unsigned 32 bit integer.
- If $i_2 = peak$ then $array[i_2 - 1] =$ the maximum value storable by an unsigned 32 bit integer.
- If $i_1 = -1$ then $array[i_1] = 0$ (the minimum value storable by an unsigned 32 bit integer).
- If $i_2 = |array|$ then $array[i_2] = 0$.

This way the binary searches can read one index outside the ranges of the monotonic arrays, while preserving the monotonic property. The reason why this is necessary is that we want to iterate on both binary searches at the same time until we satisfy the condition

$$\begin{aligned}&(array[i_1] < array[i_2] < array[i_1 + 1] \lor array[i_2] < array[i_1] < array[i_2 - 1])\\ &\land (peak - i_1 - 1) + (i_2 - peak) = k \left\lceil \frac{|array|}{32} \right\rceil\end{aligned}$$

with k being the thread index. The condition has two parts:

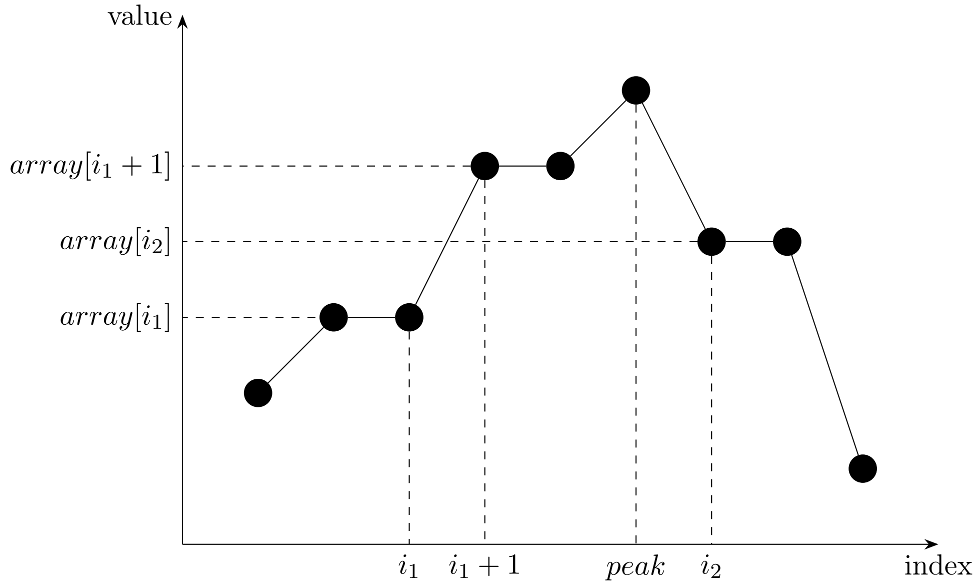

Figure 4: An example of a stabilised set of binary searches for the bitonic merge.

1. By requiring that $array[i_1] < array[i_2] < array[i_1+1]$, we force the indices to be as seen in fig. 4, where the element at index i_1 is the first element on the left side of the peak which is smaller than the element at index i_2 on the right side of the peak.
   Likewise, if $array[i_2] < array[i_1] < array[i_2-1]$ is satisfied instead, i_2 is the first element which is smaller than i_1.

2. The condition $(peak-i_1-1)+(i_2-peak) = k\left\lceil\frac{|array|}{32}\right\rceil$ has the purpose of distributing the work amongst the threads. Specifically, if we assign $\left\lceil\frac{|array|}{32}\right\rceil$ elements to each of the 32 threads, then $k\left\lceil\frac{|array|}{32}\right\rceil$ is the offset from the end of the output array at which thread k should start writing its output. $(peak-i_1-1)+(i_2-peak)$ denotes the number of elements that has already been merged, which in this case means that if we require that $(peak-i_1-1)+(i_2-peak) = k\left\lceil\frac{|array|}{32}\right\rceil$, thread k skips the largest $k\left\lceil\frac{|array|}{32}\right\rceil$ elements.

Once the two binary searches are done, thread k knows the state at which the sequential merge algorithm would have been after merging $k\left\lceil\frac{|array|}{32}\right\rceil$ elements, and it can therefore start merging the next $\left\lceil\frac{|array|}{32}\right\rceil$ elements independently of the state of the other threads.

For the sake of using the merged list to calculate the ANSV, instead of storing the values of the elements, we store the indices that the merged elements had in the input array.

Running two binary searches concurrently only performs at most $2\lg(n)$ steps, so the work and span are dominated by actually merging elements. Since the number of threads is constant, both work and span are $O(n)$.

7.4.5 Rightmost and leftmost smaller value

Another small algorithm is finding the rightmost element in an array that has a value smaller than a target value. The input is the array and the target value. We start by splitting the array into 32 parts, assigning each part to a thread which finds the index of the rightmost smaller element in the part sequentially. We then synchronize the threads, and use the tree reduction method to find the largest index amongst the threads.

We likewise find the leftmost smaller element by finding the leftmost smaller element in the part, and then finding the smallest index in the array via the tree reduction method.

This obviously runs in $O(n)$ time and with $O(n)$ span.

7.5 The ANSV algorithm

Now for the actual ANSV algorithm. First we need to note a few things regarding the explanation of the algorithm itself:

- The algorithm is split into six kernels, which we will present first, after that we will present how these kernels are executed with regards to input and output. The reason is that we need to accommodate the limitations of how the GPU synchronizes threads.
- Each kernel is analysed with regards to the work performed by a single workgroup and the span of one thread. $O(n)$ denotes linear work relative to the length of the entire array that is passed as input to the kernel, while $O(m)$ denotes linear work relative the length of the part of the array that is assigned to the specific workgroup (typically a subset).
- We assume that all the input elements are distinct, since the algorithm is based on the ANSV algorithm from (Berkman et al., 1993).

7.5.1 Kernel 1: Minimum within subset

The first kernel's job is to calculate the minimum within each subset, with the input to each workgroup being an array containing the subset. This is exactly what is explained in section 7.4.2, so like what is explained there, the work is $O(m)$ and the span is also $O(m)$.

7.5.2 Kernel 2: ANSV of subset minima

The second kernel's job is to take an array of all the subset minima found by the first kernel, and each workgroup then finds the ANSV of a single element of that array.

We do this by assigning each element to a workgroup. That workgroup then runs the rightmost smaller value algorithm described in section 7.4.5 on the part of the array that is left of the element. It then likewise runs the leftmost smaller value algorithm on the part of the array that is right of the element. This gives us the left and right matches of the element assigned to the workgroup.

As the leftmost- and rightmost smaller value algorithms both perform linear work with linear span, the kernels performs $O(n)$ work and its span is $O(n)$.

7.5.3 Kernel 3: Local ANSV within subset

The third kernel's job is to solve the ANSV problem locally within each subset.

Each workgroup is assigned a subgroup and runs the entire parallel CPU ANSV algorithm as described in section 4.3 over the 32 threads in the workgroup.

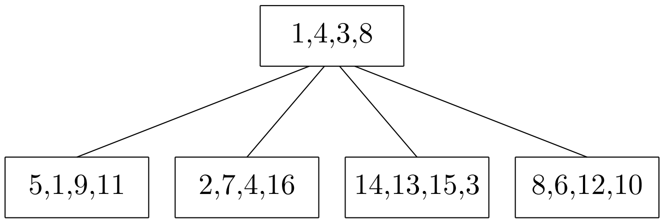

Figure 5: An illustration of the two-layer min-tree with $\sqrt{n}$ elements per node, derived from the array [5,1,9,11,2,7,4,16,14,13,15,3,8,6,12,10].

We do make a slight modification though: Instead of the min-tree being a binary tree with $\lg(m)$ layers, it is now a two-layer tree with the root node having $\sqrt{m}$ children and each node containing $\sqrt{m}$ elements. Such a tree can be seen in fig. 5.

Running the basic search procedure using the two-layer min-tree takes $O(\sqrt{m})$ time, instead of the usual $O(\log(m))$ time with the regular min-tree. But in return, we only need $m + \sqrt{m}$ space to store the two-layer min-tree, instead of the $2m$ space it takes to store the regular min-tree.

Usually, this trade-off wouldn't be worth it, but since the cache available to each workgroup is very limited, in practice the number of elements in each subset is only a few hundred. This means that the difference between $\sqrt{m}$ and $\lg(m)$ is small, while the space difference is very significant as it allows the subset size to be larger, which reduces the constant factor of time spent executing the kernel amortized over the number of elements in the subset.

The work done by the kernel is $O(m)$, since we treat the 32 threads as a constant number, and all parts of the parallel CPU ANSV algorithm performs at most linear work. The only change is the two-layer min-tree, but performing $\sqrt{m}$ work 32 times is still within $O(n)$.

As for the span of each thread, since we perform $O(m)$ work spread out over a constant number of threads, the span must also be $O(m)$.

7.5.4 Kernel 4: Global ANSV of the subset minima

Kernel 1 found the subset minima, and kernel 2 found the ANSV of the subset minima array. The job of kernel 4 is to find the nearest smaller values of the subset minima elements in the original array.

The ANSV of the subset minima array tells us which subsets contain the nearest smaller values of the subset minima elements. That is, if e.g. the minima element of subset 4, b_4, found its left nearest smaller value in the subset minima array to be b_2, then its nearest smaller value, $l(b_4)$, in the original array would be in subset 2.

The reason why this is true can be seen if we continue the example. Lets say that contradictory to our claim, $l(b_4)$ was located in subset 3. Then $b_3 \leq l(b_4) < b_4$, which means that b_3 would be the nearest smaller value in the subset minima array, not b_2.

With this knowledge, workgroup k can find the left match of b_k by letting i be left match of b_k in the subset minima array. It then runs the rightmost smaller value algorithm described in section 7.4.5 on subset i to find the left match of b_k.

Similarly, we let j be the right match of b_k in the subset minima array and find the right match of b_k using the leftmost smaller value algorithm on subset j.

Since all we do is read two values from an array and run the leftmost- and rightmost smaller value algorithms with the input array being a subset of size m, the work and span of the algorithm is $O(m)$.

7.5.5 Kernel 5: Calculate sizes of arrays to merge

Kernel 6 is tasked with executing the merging procedure, but since it is very cache-constrained, we have decided to move the calculations of what subarrays to run the merging procedure on to a separate kernel.

There are four possible pairs of subarrays to run the merging procedure on, so we denote which subarrays should be merged by writing the length of the subarrays. If the length is zero, this means that we shouldn't merge this subarray. The four pairs are:

1. Subarrays $b_i \dots l(b_k)$ and $r(b_k) \dots r(b_i)$ should be merged if $A[b_i] > A[b_j]$.
2. Subarrays $l(b_j) \dots l(b_k)$ and $r(b_k) \dots b_j$ should be merged if $A[b_i] < A[b_j]$.
3. Subarrays $l(b_k) \dots (i+1) \cdot m - 1$ and $k \cdot m \dots b_k$ should be merged if $i = k - 1$.
4. Subarrays $b_k \dots (k+1) \cdot m - 1$ and $j \cdot m \dots r(b_k)$ should be merged if $j = k + 1$.

Thread t of workgroup w handles calculating the eight subarray lengths related to b_k with $k = 32w + t$. Since all the relevant variables have already been calculated during previous kernels, this kernel performs a constant amount of calculations per thread. The work and span are therefore both $O(1)$.

7.5.6 Kernel 6: Merging subarrays

The sixth and final kernel handles finding matches across subsets via the merging procedure.

Each workgroup is assigned a set of four potential pairs of subarrays to run the merging procedure on, as calculated by kernel 5. For each pair where the length of the subarray is not zero, the workgroup knows the indices of the subarray, since it knows the start index and the length. We then use the algorithm described in section 7.4.3 to calculate the suffix minima of the left subarray and the prefix minima of the right subarray. We ensure that the suffix- and prefix minima are written to the same array, such that once we synchronize the threads, we are ready to run the bitonic merge algorithm described in section 7.4.4. The output of the bitonic merge is the elements' indices from the array containing both the suffix- and prefix minima, which means that we can know from which subarray an element originated by checking if the index is less than or greater than the length of the left subarray.

Now that we have the merged list, we can use the formulas described in the original merging procedure from section 4.2 to calculate the left and right matches of the elements in the subarrays. As in the parallel CPU ANSV algorithm, we only write the result if there has not already been found a match for the element. That is, if kernels 3 or 4 have not already found a match.

The suffix- and prefix minima algorithm and the bitonic merge algorithm all perform $O(m)$ work and their span are $O(m)$. The kernel runs these algorithms up to three times followed by calculating the indices of the matches, which is also $O(m)$ work and span. The total work performed by the kernel is therefore $O(m)$ and the span is also $O(m)$.

7.5.7 Running the entire algorithm

To round off explaining the algorithm, we will summarise what the kernels calculate, and how they are to be executed with regards to input and output.

Kernel 1 finds the subset minima. Kernel 2 then calculates the ANSV amongst the subset minima. Kernel 3 finds all the matches local to the subsets. Kernel 4 finds the ANSV of the subset minima. Kernel 5 figures out which subarrays to merge. Kernel 6 finds the remaining ANSV using the merging procedure.

The algorithm is therefore run as follows:

1. Calculate/define the subset size, m, that we split the array into.
2. Dispatch kernel 1 with $\lceil \frac{n}{m} \rceil$ workgroups, one for each subset.
   `subset_min, subset_min_index = kernel1(data)`
3. Dispatch kernel 2 with $\lceil \frac{n}{m} \rceil$ workgroups, one for each element in the `subset_min` array.
   `subset_min_ansv = kernel2(subset_min)`
4. Dispatch kernel 3 with $\lceil \frac{n}{m} \rceil$ workgroups, one for each subset.
   `ansv_partial = kernel3(data)`
5. Dispatch kernel 4 with $\lceil \frac{n}{m} \rceil$ workgroups, one for each subset.
   `subset_min_global_ansv = kernel4(data, subset_min, subset_min_ansv)`
6. Dispatch kernel 5 with $\lceil \frac{n}{m} \rceil$ workgroups, one for each subset.
   ```
   merge_indices = kernel5(data, subset_min_index, subset_min_ansv,
                           subset_min_global_ansv)
   ```
7. Dispatch kernel 6 with $\lceil \frac{n}{m} \rceil$ workgroups, one for each subset.
   ```
   ansv = kernel6(data, merge_indices, ansv_partial, subset_min_index,
                  subset_min_ansv, subset_min_global_ansv)
   ```

In practice, kernel 4 and 6 puts hard upper limits on the size of the subsets due to cache sizes, and while kernel 2 is flexible with regards to the size of its input, it becomes exponentially slower the more subsets there are. This problem can be solved, however, due to the fact that all kernel 2 does is solving the ANSV problem for a separate array. We can therefore pick the subset sizes to optimize for the cache sizes available to kernel 4 and 6, while we replace the call to kernel 2 with a recursive call to the entire algorithm. Once the recursion causes the number of subsets to be sufficiently small, we call kernel 2 in order to end the recursion.

7.5.8 Correctness

The correctness of the algorithm follows from the fact that it is just a reimplementation of the CPU version, but split into smaller pieces and with some parts replaced with versions that are more suitable for running on a GPU, but that calculate the same output.

The CPU version starts by finding the ANSV local to the subsets, the subset minima, and the prefix- and suffix minima of each subset.
In the GPU version, kernel 1 finds the subset minima, kernel 2 finds the ANSV local to the subsets, and kernel 6 calculates the prefix- and suffix minima.

The CPU version then uses the basic search procedure to find $l(b_k)$ and $r(b_k)$, and then calculates $gl(b_k)$ and $gr(b_k)$.
The GPU version uses kernel 3 to find $gl(b_k)$ and $gr(b_k)$, and then uses that in kernel 4 to find $l(b_k)$ and $r(b_k)$.

Finally, the CPU version determines which subarrays to run the merging procedure on, and then runs the merging procedure on said subsets to find the remaining matches for the ANSV output.
In the GPU version, kernel 5 calculates the subarrays to run the merging procedure on, after which kernel 6 runs the merging procedure and writes the ANSV output.

As can be seen from the above, the CPU and GPU versions of the algorithm calculate the same information, they just do it in different ways and in a different order. Therefore, since the input to the merging procedure and the merging procedure itself are the same, along with the both version calculating the same ANSV local to the subsets, the resulting output of the algorithms must also be the same, proving the correctness of the GPU version of the algorithm.

7.5.9 Work and span

Work

When we analyse the algorithm in terms of work, we need to consider both the work performed by a single workgroup when running a kernel, and how many workgroups are dispatched.

The work performed by each kernel is:

- Kernel 1 dispatches $\lceil \frac{n}{m} \rceil$ workgroups performing $O(m)$ work each resulting in $O(n)$ total work.
- Kernel 2 dispatches $\lceil \frac{n}{m} \rceil$ workgroups, and with the input array being $\lceil \frac{n}{m} \rceil$ long, each workgroup performs $O\left(\frac{n}{m}\right)$ work resulting in $O\left(\left(\frac{n}{m}\right)^2\right)$ total work.
- Kernel 3 dispatches $\lceil \frac{n}{m} \rceil$ workgroups performing $O(m)$ work each resulting in $O(n)$ total work.
- Kernel 4 dispatches $\lceil \frac{n}{m} \rceil$ workgroups performing $O(m)$ work each resulting in $O(n)$ total work.
- Kernel 5 dispatches $\lceil \frac{n}{m} \rceil$ workgroups performing $O(1)$ work each resulting in $O(\frac{n}{m})$ total work.
- Kernel 6 dispatches $\lceil \frac{n}{m} \rceil$ workgroups performing $O(m)$ work each resulting in $O(n)$ total work.

We notice that all kernels, except for kernel 2, perform $O(n)$ work. If we then replace kernel 2 with a recursive call, as mentionad in section 7.5.7, and only terminate the recursion once n is less than a constant, then kernel 2 performs constant work, and we instead get the recurrence relation $T(n) = T\left(\frac{n}{m}\right) + cn$, which describes the total work performed by the algorithm.

If we then assume the subset size to be $m = \lg(n)$, we can show that the GPU algorithm is work optimal, by guessing that the work is $T(n) \leq 2cn$ and plugging that into the recurrence.

Base case $(n = 4)$

$$\begin{aligned} T(4) &= c && \text{by definition} \\ 2cn &= 8c \geq c = T(4) = T(n). \end{aligned}$$

Induction step

Assume $T(\ell) \leq 2c\ell$ for $4 \leq \ell < n$.

$$\begin{aligned} T(n) &= T\left(\frac{n}{\lg(n)}\right) + cn \\ &\leq 2c\frac{n}{\lg(n)} + cn \\ &= cn\left(\frac{2}{\lg(n)} + 1\right) \\ &\leq 2cn = O(n). \end{aligned}$$

Span
If we instead look at the span of each kernel we see that the total span is $O(m)$ except for kernel 2.

- Kernel 1 has each workgroup process a length m subset, so its span is $O(m)$.
- Kernel 2 has each workgroup process a length $\lceil \frac{n}{m} \rceil$ subset, so its span is $O\left(\frac{n}{m}\right)$.
- Kernel 3 has each workgroup process a length m subset, so its span is $O(m)$.
- Kernel 4 has each workgroup process two length m subsets, so its span is $O(m)$.
- Kernel 5 only performs a constant number of calculations, so its span is $O(1)$.
- Kernel 6 has each workgroup merge a constant number of length m subsets, so its span is $O(m)$.

If we then again replace kernel 2 by a recursive call, we get the recurrence $T(n) = T\left(\frac{n}{m}\right) + cm$. We then again assume that the subset size is $n = \lg(n)$ and guess that the span is $T(n) \leq c\lg^2(n)$ which we plug into the recurrence:

Base case $(n = 4)$

$$\begin{aligned} T(4) &= c && \text{by definition} \\ c\lg^2(n) = c\lg^2(4) &= 4c \geq c = T(4) = T(n) \end{aligned}$$

Induction step
Assume $T(\ell) \leq c\lg^2(\ell)$ for $4 \leq \ell < n$.

$$\begin{aligned} T(n) &= T\left(\frac{n}{\lg(n)}\right) + c\lg(n) \\ &\leq c\lg^2\left(\frac{n}{\lg(n)}\right) + c\lg(n) \\ &= c\left(\lg^2\left(\frac{n}{\lg(n)}\right) + \lg(n)\right) \\ &= c\left((\lg(n) - \lg(\lg(n)))^2 + \lg(n)\right) \\ &= c\left(\left(\lg^2(n) - 2\lg(n)\lg(\lg(n)) + \lg^2(\lg(n))\right) + \lg(n)\right) \\ &= c\lg(n)\left(\lg(n) - 2\lg(\lg(n)) + \frac{\lg^2(\lg(n))}{\lg(n)} + 1\right) \\ &\leq c\lg^2(n) = O(\log^2(n)). \end{aligned}$$

The span of the algorithm is therefore $O(\log^2(n))$ if we set the subset size to $\lg(n)$ and use $\frac{n}{\lg(n)}$ threads.

8 Performance optimizations and comparisons

In this section we will discuss implementation, optimization and performance comparisons between the algorithms.

8.1 Implementation

For our benchmarks, we prepared three implementations of the ANSV algorithm available.

1. The original CPU algorithm from (Berkman et al., 1993).
2. Our modified CPU algorithm that handles duplicate values.
3. The BSZ CPU algorithm from (Shun & Zhao, 2013b).
4. Our GPU algorithm.

We started by implementing the CPU algorithm from (Berkman et al., 1993). We picked the Rust programming language, since it is fast and because the Rust compiler checks for concurrency bugs at compile time (instead discovering them at runtime), making it a great programming language for implementing parallel algorithms. In addition, we used the Rayon library to handle thread pools and distributing work across threads. This implementation was compiled with Rust's `--release` option.

We also easily implemented our modified CPU algorithm by modifying the implementation of the original CPU algorithm.

We found the BSZ implementation used in (Shun & Zhao, 2013b) on GitHub.[4] This implementation is written in C++ and uses OpenMP to implement their parallelism. It was compiled with GCC's `-O2 -DOPENMP` options.

When implementing GPU algorithms in algebraic research, the conventional choice is to use CUDA, since Nvidia (the makers of CUDA) has the best support and tooling in the industry. We did, however, have access to a mix of Nvidia and AMD GPUs, with the most accessible GPU being an AMD GPU. With CUDA only being supported on Nvidia GPUs, this was no longer the optimal choice. Instead, we decided to target Vulkan, which is an open API that works on both Nvidia and AMD GPUs. For code to run via Vulkan, it needs to be compiled to SPIR-V which is an intermediary language. We decided to use the RustGPU project, which compiles Rust code to SPIR-V kernels. While RustGPU is admittedly still very early in debelopment and buggy, it provides the benefit of allowing us to reuse parts of our codebase, since everything is written in Rust. To execute the SPIR-V kernels, we used wgpu which is a library based on the WebGPU API that support running kernels via Vulkan.

This implementation was compiled with Rust's `--release` option.

One important difference between the implementation are the data types. The BSZ implementation assumes the input is signed 32 bit integers. Our implementations of the CPU algorithms are made to accept any data type as input, but we tested them with signed 32 bit integers to make it as comparable as possible.

For our GPU implementation, however, we decided to let the input be unsigned 32 bit integers, since it made a complex codebase slightly simpler.

[4]Shun and Zhao, 2013a.

8.2 Test systems

The hardware that we had available for our tests was a mix of our personal hardware and the hardware on DTU's HPC cluster.

Our personal hardware includes a six core Ryzen 5 7600x CPU, an AMD Radeon RX 7900XTX GPU.

For the systems on DTU's HPC cluster we used

- A system with a 12 core Xeon E5-2650 v4 CPU.
- A system with a 16 core Xeon Gold 6226R CPU.
- A system with a Nvidia Tesla V100 GPU.
- A system with a Nvidia Tesla A100 GPU.

This means that we have three CPUs and three GPUs available for our tests. It should be noted that this hardware was released up to six years apart (the Xeon E5-2650 v4 is from 2016, and the Ryzen 5 7600x is from 2022), so we need to be cautious if we compare benchmarks across different hardware.

All the systems were Linux systems, with the systems with the Xeon CPUs and the Nvidia GPUs running RHEL Linux on kernel 6.1 LTS, and our personal system running Arch Linux on kernel 6.15.3.

The Nvidia GPUs were using Nvidia's proprietary driver.

The AMD GPU was tested twice, once with the open source RADV Vulkan driver, and the other with the proprietary AMDGPU Pro driver.

8.3 Test setup

When deciding on which data we would run the benchmarks on, we made four observations:

- Performing the merging procedure is a significant part of the computations done, so to present a worst case scenario, we want the subarrays that are merged to be as long as possible.
- If we make the input too regular, it becomes easier for the CPU's and GPU's branch prediction and memory prefetching hardware to optimise the code execution.
- It is obvious, that with a small dataset the CPU algorithms will be faster, since dispatching GPU kernels is slow, and the main benefit of a GPU is that it has many cores that can parallelise over large datasets.
- When connecting to a GPU, the WebGPU API includes a process for negotiating which limits are supported. One such limit is the maximum buffer size, which is the maximum size of an array that we can use as an argument for a kernel. With our hardware, the maximum supported limit is approximately 536 million 32 bit elements.

For these reason we decided on two datasets.

The first dataset is an array of 500 million elements constructed in order to maximise the time spent running the merging procedure. Specifically, we arrange the elements such that if we iterate over the elements from smallest to largest, we would alternatively pick a value from the start of the array and the end of the array. An array of six elements would therefore look like [0,2,4,5,3,1].

Since it was designed to potentially force a worst case runtime, we refer to this as the "worst case" dataset.

The second dataset is an array of 500 million random distinct elements. The purpose of this dataset is to not be biased in any way, and to serve as a reference point compared to our first dataset. We refer to this as the "random" dataset.

For the tests themselves, we ran each algorithm five times to warm up the caches followed by 30 runs of which we saved the average times.

Specific to the CPUs, since we had a mix of 6, 12, and 16 core CPUs, and each CPU core implements multithreading (having one core run two threads at once), we decided to get data points at 1, 6, 12, 16, 24, and 32 threads, with each CPU being tested up to twice its core count.

Specific to the GPUs, we had to make a choice regarding what we include in the benchmark time. Since we both generate the data and read the output on the CPU, one could argue that the GPU algorithm needs to include transferring the data to the GPU and transferring the output back to the CPU. This makes sense since eventually, the data would start and end at the CPU. We, however, would argue that transferring data back and forth does not belong in the benchmark. Our argument is that while it is true that the data starts and ends on the CPU, our algorithm is supposed to run in the larger context of a full compression algorithm. So while transferring data back and forth is part of the actual runtime, it would be amortized between all parts of the compression algorithm, and we have no way of knowing how much of the runtime would be spent on the ANSV algorithm. We therefore do not include the time spent transferring data in our benchmarks.

8.4 Results

The data that we collected can be see in table 1, table 2, table 3, table 4, table 5, table 6, and table 7 in the appendix. For the CPU benchmarks, we have split the data by the input dataset and visualised it in fig. 6 and fig. 7.

The patterns that we see in the figures are:

- Our implementations are slightly slower but overall similar to the performance of the BSZ implementation.
- When the CPUs start using multithreading (i.e. when they spawn more threads than there are cores on the CPU), the BSZ implementation using OpenMP continues to speed up, while our implementation using Rust and Rayon stops scaling.
- Our implementation of the original CPU algorithm is slightly slower than our modified algorithm.

  While our modified algorithm was targeted at handling generalised data, our modified merging procedure also has the added benefit of not needing to calculate the prefix- and suffix minima of the data. This, along with the merging procedure maybe being more efficient are the only two *specific* parts of the changes that we can point to which would cause a speed-up. The other changes (notably the new logic for which subarrays to run the merging procedure on) are too complex and situational for us to know if they even affect the runtime practice.
- Overall, the algorithms ran faster on the worst case dataset than they did on the random dataset.

  This suggests that we were right in worrying about the branch prediction and memory prefetching hardware being very efficient at tackling the worst case dataset.

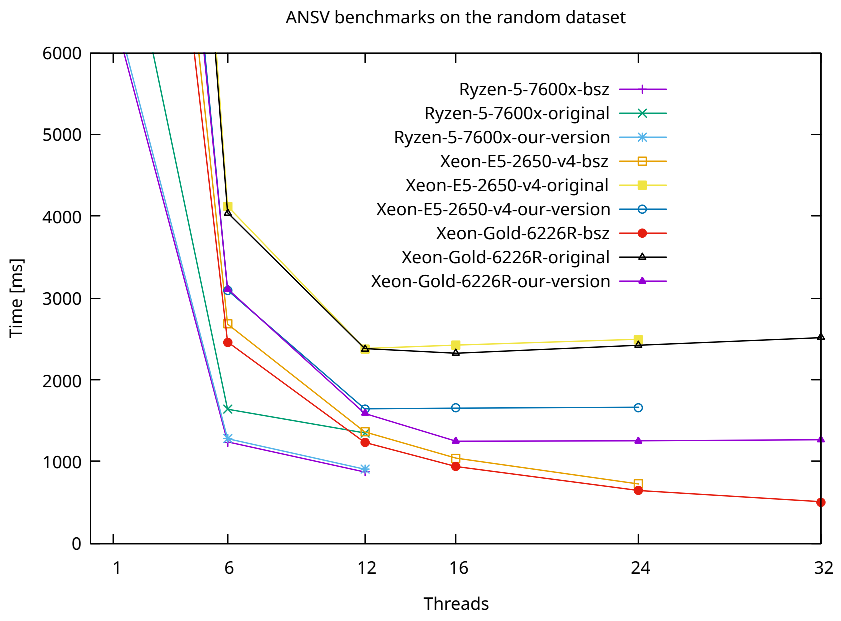

Figure 6: A plot of the data from table 1, table 2, and table 3. To not clump the lines too much, we have cut off the data points for when the benchmarks are run on one thread.

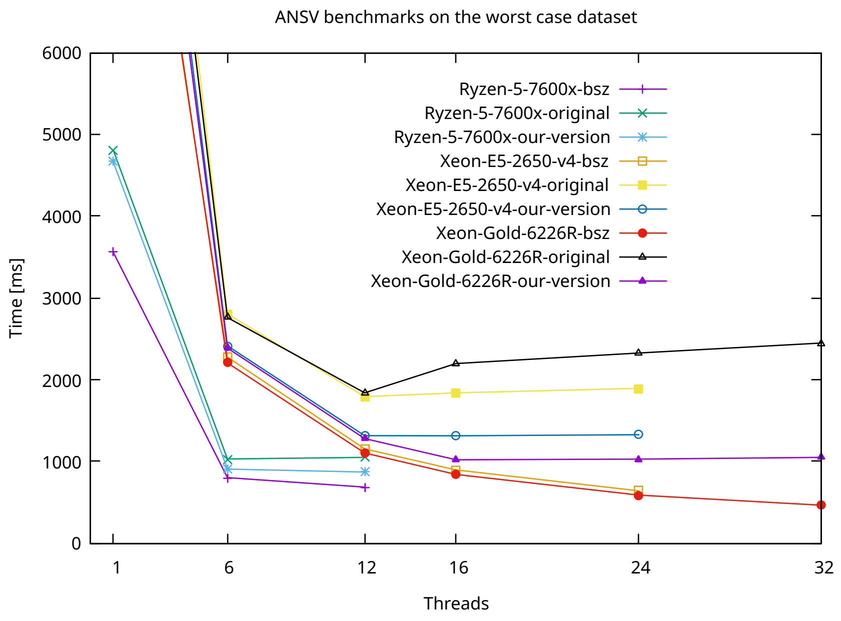

Figure 7: A plot of the data from table 4, table 5, and table 6. To not clump the lines too much, we have cut off the data points for when the benchmarks are run on one thread.

Overall, it seems like our modified CPU ANSV algorithm is competitive in terms of performance, although at the cost of being more complex compared to the other algorithms. Where is really shines is that it does not need to run twice in order to handle data with duplicate values.

The fastest benchmark run of any CPU implementation was the BSZ algorithm on the Xeon Gold 6226R CPU. This finished in 510ms on the random dataset and in 466ms on the worst case dataset.

If we compare it to the GPU benchmarks in table 7, we see the fastest GPUs are faster, with the fastest being the AMD Radeon RX 7900 XTX (using the RADV driver) which finished in 98ms on the random dataset and 118ms on the worst case dataset. If we compare all the data, we make the following observations:

- The GPUs are ranked in speed as RX 7900XTX > Tesla A100 > Tesla V100, which coincides with them being released in 2022, 2020, and 2017 (in the same order).
- The RX 7900XTX has huge differences in performance between which driver is used, with the RADV driver being 4-5 times faster than the AMDGPU Pro driver. What this suggests is that the Nvidia GPUs might also have potential for some additional large performance optimisations.
- In the GPU benchmarks, the worst case dataset was slower than the random dataset (with one exception), which is the opposite as what we saw for the CPU benchmarks.

  This suggests that the CPUs are more reliant on either branch prediction or memory prefetching, compared to GPUs.

We would also like to point out that since the Xeon Gold 6226R was released in 2020, the AMD Radeon RX 7900XTX in 2022, and the Tesla A100 in 2020, this performance difference between the CPU and GPU algorithm is not a result carried by one set of hardware being significantly newer (and therefore faster) than the other. Similarly, the Xeon CPU has an Manufacturer Suggested Retail Price (MSRP) of $1500, while the AMD GPU has a MSRP of $1000, so it also is not a problem of some hardware being “a higher tier” of product than the rest.

8.5 Optimising the CPU algorithm

There is one significant difference between the description of the ANSV algorithms and our implementations. For the analysis of both ANSV the original ANSV algorithm and our modified version, it is optimal to have the subset size be $\lg(n)$, such that we can utilize $\frac{n}{\lg(n)}$ threads. In practice, this results in very poor performance since $\lg(n)$ is tiny, so the overhead per thread is large, and we do not even have hardware capable of anything close to $\frac{n}{\lg(n)}$ threads simultaneously. Instead, we put a minimum on the subset size, so it is at least 8192 elements.

After fixing that, our preliminary testing showed that our CPU implementations were slightly slower than the BSZ implementation. When we tried profiling our code to find potential speed-ups, the profiling tools claimed that most of the time was spent during Linux system calls and during codepaths of the Rayon library. In other words, it was hard to figure out what the bottlenecking code was. For that reason, we tried replacing the Rayon library with a manual handling of threads using the `std::thread` library from Rust’s standard library, but this just resulted in the code running much slower. We also tried just manually looking for places to optimise our code, but we didn’t have much luck.

So while our code is slightly slower than the BSZ implementation, we believe it is mostly related to our choice of algorithm, programming language, and (if we believe the profiling output) library selection.

8.6 Optimising the GPU algorithm

We had a better time optimising the GPU algorithm. As the GPU we had the most readily available was an AMD Radeon RX 7900XTX we optimized for that GPU. We used the RadeonGPUProfiler software to visualize the time spent during different parts of the algorithm. Sadly, this software only runs using the AMDGPU Pro diver, so we do not include profiling of the RADV driver. As seen in fig. 8, the majority of the time is spent running kernel 3 and 6, so we focused our optimization efforts there.

Additionally, there is also a lot of time spent seemingly doing nothing without any indication of what is happening. Based on our experience of looking at the profiling output of different GPU algorithms, and the fact that during the recursive part of the algorithm between kernel 1 and 3 the time spent doing nothing is relatively small, we guess that it might be related to synchronizing large amounts of workgroups and writing their outputs to memory, although we have no way to confirm or deny our guess. It might also just be an artefact of the profiling process.

The first thing we notice regarding kernel 3 and 6 is that the yellow bar is very small. The yellow bar signifies how "busy" the GPU is, that is how many workgroups are currently being processed at a time. There is, however, the small detail that what it actually describes is how many workgroups are loaded in the SIMD's cache. As we mentioned in section 7.2.2, each SIMD loads multiple workgroups into its cache, but only processes one of them at a time. This means that the yellow bar being low is a sign of being cache constrained, but not necessarily a sign that the hardware is inactive. But while the hardware might not be inactive, it is still a sign that we are cache constrained, so we focus our effort on optimising the cache usage of kernel 3 and 6.

We first tried to just write better code. This is what caused kernel 3 to implement the min-tree using a two-layer tree instead of a binary tree (see section 7.5.3). The second step was to optimise the subset size. Just like with the CPU algorithm, we do not have hardware capable of running anything close to $\frac{n}{\lg(n)}$ threads simultaneously, so our initial implementation was set to have a large subset size. What this resulted in was the GPU crashing. After some testing, we found that using the AMDGPU Pro driver runs out of cache (and crashes) if we set the subset size to anything higher than 1000 elements. We then tried different subset sizes, and found anything aroudn 400 to be optimal.

While the number 400 is not special we can still reason why anything close to it would be optimal. For smaller workgroup sizes, we need to remember that in kernel 6 we run two binary searches per thread. Since the input to the kernels is two subarrays, the binary search uses up to $\lceil \lg(2 \cdot 400) \rceil = 10$ steps, which means that each thread performs at least something multiplied by 10 operations. When we then split the 800 elements between the 32 threads in a workgroup, we see that $\lceil \frac{800}{32} \rceil = 25$ which means that each thread only calculates 25 elements of the output. If the subset size was any smaller, the time spent running the binary search amortized over the number of output elements per thread would be even higher.

The reason why we do not increase the workgroup size beyond 400 is because of another thing we mentioned in section 7.2.2: That multiple SIMDs share the same LDS cache. So while we *could* increase the workgroup size, it would result in some SIMDs not having enough cache space to store its input. Those SIMDs would therefore do nothing, resulting in actual hardware sitting idle (unlike when we just couldn't load multiple workgroups into each SIMD).

The third thing we tried, was to replace the RADV driver with the AMDGPU Pro driver. While the profiling tool only supports the AMDGPU Pro driver due to both being made by AMD, the RADV driver is made by the community and it has a reputation of being much better, so it was an obvious last thing to try. We first found that the RADV only crashes when the subset size becomes 2500 elements. This is 2.5 times larger than with the AMDGPU Pro driver. We still, however, found the optimal subset size to be around 400. While we can't confirm it due to the profiling tools not supporting and RADV driver, this increase in the maximum subset size suggests that the RADV driver is somehow much more cache-efficient compared to the AMDGPU Pro driver. But whether it is due to cache-efficiency or not, the data in table 7 clearly shows a massive speed-up by switching drivers.

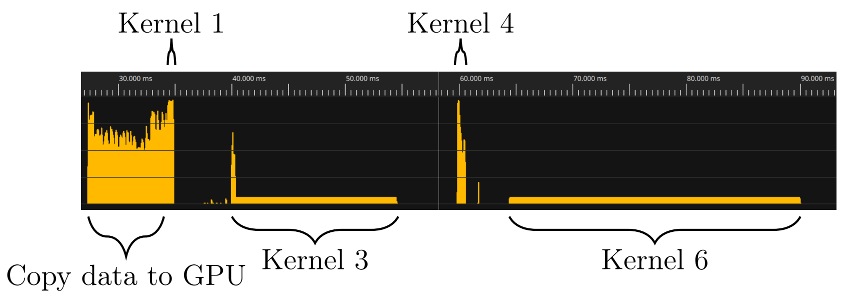

Figure 8: A screenshot of part of the output when profiling the GPU ANSV algorithm using the RadeonGPUProfiler software. The horizontal axis represents the time spent, and the vertical axis represents how "busy" the GPU is measured as the number of workgroups actively running on the hardware. The unmarked parts are the recursive call replacing kernel 2 in-between kernel 1 and 3, and kernel 5 which is inbetween kernel 4 and 6.

Part II
Range Minimum Query

Now that we have an efficient GPU algorithm for solving the ANSV problem, it would be interesting to see if we could utilize it to solve another problem. One candidate for such a problem is the range minimum query problem (henceforth RMQ). RMQ is used in the parallelised Lempel-Ziv-77 algorithm[5], and it is mentioned in (Berkman et al., 1993) that the ANSV of an array can be used to solve RMQ.[6]

Definition 8.1 (Range Minimum Query)**.** Given an array A of n integers, answer questions of the form $\text{RMQ}(i, j) = \arg\min A[k]$ for $i \leq k \leq j$. In words, find the minimum element in the range $[i; j]$ of A.

The way to solve RMQ via ANSV is by turning it into a Cartesian tree, and then answering Lowest Common Ancestor queries (henceforth LCA) in the Cartesian tree.[7]

Definition 8.2 (Cartesian Tree)**.** Given an array A of n distinct numbers, the Cartesian tree of A is defined using the following recursive procedure:

1. Find the smallest element in A. This is the root element, r.
2. Split A into two subarrays. The left subarray is $A[0 \ldots r-1]$. The right subarray is $A[r+1 \ldots n-1]$.
3. Construct the Cartesian tree of both subarrays.
4. Place an edge between r and the root of the Cartesian tree constructed from the left subarray.
5. Likewise, place an edge between r and the root of the Cartesian tree constructed from the right subarray.

Definition 8.3 (Lowest Common Ancestor)**.** Given a tree T and two nodes a and b. The lowest common ancestor of a and b is the shared ancestor of a and b which is the furthest away from the root of T.

It is very easy to convert ANSV into a Cartesian tree, and it is also trivial to parallelise. All we have to do it iterate over the ANSV array, and for each element, pick whichever of its left and right nearest smaller values that is the largest.

The question therefore becomes how to find the LCA in constant time.

9 Answering LCA in constant time

We have decided to use the method described in (Schieber & Vishkin, 1988) to solve the LCA problem. For proof that it works, and an explanation of why, we refer to the article. The article describes a preprocessing stage where four values are calculated for each node in the Cartesian tree. Along with a lookup table, these four values enables answering LCA queries in constant time.

The four precomputed values are:

[5]Shun and Zhao, 2013b, p. 126.
[6]Berkman et al., 1993, p. 364.
[7]Berkman et al., 1993, p. 364.

- **Preorder**: This is simply the index of the node in the preorder traversal of the tree, with the root node having index one.
- **Level**: This is the node's distance from the root node.
- **Inlabel**: This is whatever preorder value amongst the node itself and its children that has the most consecutive zeroes in its binary representation counting from the right side.
  So if a node has e.g. preorder value 3 and it has 4 children, then the preorder of those would be 3 through 7 and we would look at the values 011, 100, 101, 110, and 111, of which 100 has the most consecutive zeroes from the right side. Node 3's inlabel would therefore be 100.
- **Ascendant**: This value stores the inlabel of all of its ancestors in a single integer. It does this by taking the rightmost one-bit from each of its ancestors (including itself), and otherwise letting all other bits be zero.

The lookup table is called **head** and it maps each inlabel value to whichever node containing said inlabel has the lowest level. Due to the inlabel being the preorder value of a descendant node, the nodes sharing the inlabel value form a path, meaning that "whichever node containing said inlabel has the lowest level" is a single unique node.

9.1 How to parallelise the precomputation

Parallelising computing all of the four precomputed values relies on parallelising the list ranking problem.

Definition 9.1 (List Ranking)**.** Given a linked list, determine the rank of each element in the list, that is the distance from the first element.

For now, we assume that we already have a parallel list ranking algorithm. The list ranking problem has a set of implicit weights, in that the distance between each element is one. We make these weights explicit by assigning a weight to each element denoting the distance to the next element.

The tree that we receive is defined by each node having an edge to its parent. The first thing we do is convert this tree to an euler tour, by adding an edge in the opposite direction. We then define two arrays, W_1 and W_2, which contains the weights of edge. $W_1[e] = 1$ if e is directed away from the root node, while $W_1[e] = 0$ when it is directed towards the root node. $W_2[e] = 1$ if e is directed away from the root node, while $W_2[e] = -1$ when it is directed towards the root node.

We then run the list ranking algorithm twice using these weights to get the distances D_1 and D_2. The preorder of a node, u, is the first distance of the edge from u to its first child, v, so $preorder[u] = D_1[u \to v]+1$. The level is the second distance to the first child, so $level[u] = D_2[u \to v]$. If we let p be the parent node of u, we can calculate the inlabel as $inlabel[u] = 2^i \left\lfloor \frac{D_1[u \to p]+1}{2^i} \right\rfloor$ with $i = \lfloor \lg(D_1[u \to v] \operatorname{xor}(D_1[u \to p] + 1)) \rfloor$ (with xor being the bitwise xor operator).

We now define a new set of weights W_3 where, for each node that is not the root node and where $inlabel[u] \neq inlabel[p]$, then $W_3[p \to u] = 2^i$ and $W_3[u \to p] = -2^i$, with i being the index of the rightmost one-bit in $inlabel[u]$. The weights of all other edges are zero. We run the list ranking algorithm with W_3 and find the distances D_3. We can now calculate the ascendant values as $ascendant[u] = D_3[u \to v] + 2^l$ with l being the number of bits required to store the number of nodes in the input tree.

The final thing we need to calculate is the head table. We do this by having each node write $head[inlabel[u]] = u$ if $inlabel[u] \neq inlabel[p]$, or if it is the root node. This is trivial to parallelise as all the calculations are independent of each other.

9.2 How to answer LCA queries

As we mentioned earlier, the nodes that share the same inlabel value form a path. The logic behind answering LCA queries relies on this fact, as what we do is first find he inlabel value of the LCA, and then figuring out exactly which node is the LCA.

When we try to answer the LCA query for two nodes x and y, the first thing we check is if they share the same inlabel value. If that is the case then one of them must be the ancestor of the other, so the LCA must be whichever one has the lowest level value. If it is not the case, we try to determine the inlabel of the LCA, which we will refer to as z.

We remind the reader that l is the number of bits required to store the number of nodes in the input three. This also means that l bits is large enough to store either of preorder, level, inorder, and ascendant for any given node, which is why we assume that all the values are l bit integers. It is find in practice to use more than l bits if we just pad the leftmost bits with zeroes, in which case the reader should replace any reference to e.g. $l - i$ with the number of bits used minus i. Back to the explanation.

We first let i be the maximum of the following three indices:

- The index of the leftmost bit in which $\mathit{inlabel}[x]$ and $\mathit{inlabel}[y]$ differ.
- The index of the rightmost one-bit in $\mathit{inlabel}[x]$.
- The index of the rightmost one-bit in $\mathit{inlabel}[y]$.

We then let j be the index of the rightmost one-bit in amongst the leftmost $l - i + 1$ bits of $\mathit{ancestor}[x] \,\&\, \mathit{ancestor}[y]$ (with & being the bitwise and operator). $\mathit{inlabel}[z]$ is then the $l-j$ leftmost bits of $\mathit{inlabel}[x]$ followed by one one-bit and j zero-bits.

The next step is for x (and y) to find its lowest ancestor which has the same inlabel value as z, named $\widehat{x}$. We do this by letting k be the leftmost one-bit in the rightmost j bits of $\mathit{ascendant}[x]$. $\widehat{x}$ is then the parent of the head of the path of nodes with their inlabel being the $l - k$ leftmost bits of $\mathit{inlabel}[x]$ followed by one one-bit and k zero-bits. It is also possible that x (or y) already has the same inlabel as z, in which case we skip this calculation and let $\widehat{x} = x$ directly.

Once $\widehat{x}$ and $\widehat{y}$ have been found, z, that is the LCA of x and y, is just whichever one of them has the lowest level value.

To better understand this process, lets look at fig. 9, which is an example of a tree with its precomputed values. If we want to find the LCA of node 3, x, and node 6, y, (named by their preorder value), we start by seeing that they do not share an inlabel value. Therefore we calculate the following:

- $i = 2$, since $\mathit{inlabel}[x]$ and $\mathit{inlabel}[y]$ differ in their third least significant bit.
- $j = 2$, since the third least significant bit is the rightmost one-bit in $\mathit{ancestor}[x] \,\&\, \mathit{ancestor}[y] = 011 \,\&\, 110 = 100$.
- Regarding $\widehat{x}$, the leftmost one-bit amongst the rightmost j bits of $\mathit{ascendant}[x]$ is the bit with index 0, so $k = 0$. We therefore look at the head node of the inlabel path with value 011, which is x itself. Its parent is node 2, which is $\widehat{x}$.
- Regarding $\widehat{y}$, the leftmost one-bit amongst the rightmost j bits of $\mathit{ascendant}[x]$ is the bit with index 1, so $k = 1$. We therefore look at the head node of the inlabel path with value 110, which is node 5. Its parent is node 1, which is $\widehat{y}$.
- Since, $\mathit{level}\,[\widehat{x}] = 1 > 0 = \mathit{level}\,[\widehat{y}]$, then $\widehat{y}$ (i.e. node 1) must be the LCA of node 3 and node 6.

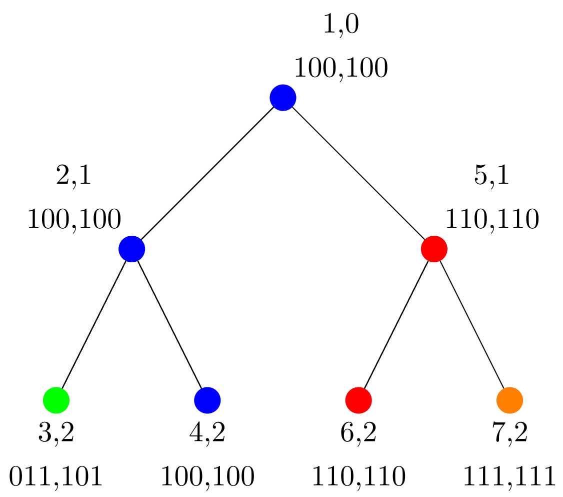

Figure 9: An example of a tree, with each node having the four values preorder, level, inlabel, and ancestor. Inlabel and ancestor are given in binary form. Each inlabel path is given a colour.

9.3 Parallel list ranking

To solve the list ranking problem in parallel, and with an algorithm suitable for a GPU, we found Rehman et al., 2009 which describes the Recursive Helman-JáJá algorithm.

To explain the Helman-JáJá algorithm, it is easier to first look at a parallel prefix sum algorithm.

9.3.1 Parallel prefix sum

The prefix sum calculation can be solved in four easy steps.

1. We split the array into equal sized parts, one for each thread.
2. Each thread calculates the sum of the elements in its part.
3. One thread calculates the prefix sum of the part sums.
4. Thread k calculates the prefix sum of part k, but instead of initializing the prefix sum calculation with zero, it initializes it with the k'th value from the prefix sum of the part sums. This effectively adds the sum of all the elements earlier in the array to the prefix sum of the part itself.

To parallelise the algorithm further, step 3 can be replaced with calling the algorithm recursively.

9.3.2 Helman-JáJá

The concept behind the Helman-JáJá algorithm is the same as that of the parallel prefix sum algorithm.

1. Split the linked list into smaller parts.
2. Calculate the sum of each part.
3. Construct a linked list of the part sums.
4. Solve the list ranking problem for the part sums.

5. Solve the list ranking problem for each part, but add the rank of the part sum to all the elements.

The difference here is that since the input is a linked list instead of an array, it is no longer possible to split the list into equal sized parts. What we do instead is to randomly decide on elements that form the start of each part. We then assign a thread to start at each start-element, and as it follows the linked list to calculate the prefix sum of its part, when it encounters another start-element, it stops.

This is a way to split the linked list into parts, but it has two downsides. The first downside is that we cannot rely on randomly picking the first element of the linked list to be a start-element. There are multiple ways to solve this problem, but in our case, we rely on the fact that it is easy to find the last element of the list, since it is the only element that does not point to the list, and that since the input is an Euler tour that we construct ourselves, we can just cleverly construct it such that we can find the first element given the last element. This will be explained later.

The second downside is that fact that we do not have a guarantee for the length of each part of the linked list, and it might therefore end up running in linear time despite our best efforts.

10 LCA on a GPU

Now that we have a method for answering LCA queries, we will start describing the individual parts necessary for an GPU implementation, after which we will assemble them together into an algorithm.

10.1 Kernel 1: Convert ANSV into a Cartesian tree

To convert the ANSV output into a Cartesian tree, all we have to do is to have each element point to whichever of its nearest smaller values has the largest value. For this we require the input to be the indices of the left and right matches from the ANSV algorithm along with the original array itself, so that we can read the values at those indices.

Since each element is independent of each other, we will not group the threads by workgroups. Instead thread t in workgroup k calculates its global index, that is $32k + t$, and handles that element. Henceforth, whenever we refer to threads' global indices, it is because their calculations are independent of each other, so we ignore their workgroups.

This kernel obviously performs $O(n)$ work and has $O(1)$ span.

10.2 Kernel 2: Count children part 1

All Cartesian trees share two properties:

- Each node has at most two children.
- If a node has two children, one child is to the left of the node and the other is to the right of the node.

Since each child of a node has the edge to the node, it knows both its own index and the index of the node. It therefore knows if it is the left child or the right child of the node.

This kernel sets a flag for each child of a node, which enables us to count the number of children. The input is the Cartesian tree.

We initialise an array of zeroes containing two elements per node in the Cartesian tree. Thread t (global index) checks if node t is the left or right child of its parent, p. If it is the left child, it writes a one to index $2p$. If it is the right child it writes a one to index $2p + 1$.

This obviously performs $O(n)$ work and has $O(1)$ span.

10.3 Kernel 3: Count children part 2

This kernel counts the number of children based on the information provided by kernel 2.

The input is the array $childflags$ produced by kernel 2.

We initialise an array of the same size as the Cartesian tree. Thread t (global index) calculates $childflags[2t] + childflags[2t+1]$ and writes it to index t in the output array. This array now contains the number of children of each node.

This obviously performs $O(n)$ work and has $O(1)$ span.

10.4 Kernel 4: Uniform add

An uniform add kernel is a kernel that adds a predetermined value to all elements in an array.

For our purpose, it is convenient to implement it such that instead of the value being a single constant, we let each workgroup add an unique value to the elements in its assigned part of the array.

The input is therefore the data array and an array containing the values to add. The size of the part assigned has so far been given implicitly to all kernels, so we do not consider it part of the explicit input.

This kernel obviously performs $O(m)$ work and has $O(m)$ span.

10.5 Kernel 5: Workgroup-local prefix sum

To implement a prefix sum algorithm on a GPU as described in section 9.3.1, all we need is a kernel that calculates the prefix sum of individual parts of an array, along with an uniform add kernel (as described previously).

For this kernel, workgroup k is assigned part k which it splits into 32 subparts. Thread t in the workgroup then calculates the sum of subpart t and saves it to index t in a small 32-element long array shared only locally between the threads in the workgroup. The threads then synchronize. Thread t then calculates the sum of sums 0 to $t-1$, and uses the result as the initial value for calculating the prefix sum of subpart t.

The input to this kernel is an array of data.

This kernel obviously performs $O(m)$ work and has $O(m)$ span.

10.6 Kernel 6: Convert Cartesian tree to Euler tour

This kernel has the task of converting an array containing a Cartesian tree into an array containing an Euler tour of said tree.

The properties we mentioned in kernel 2 are also essential here in order to deterministically create an array describing an Euler tour through a Cartesian tree.

The array containing the Cartesian tree is actually a set of edges, since if the value at index u is v, then there must be an edge $u \to v$. For an Euler tour of a tree, this means that each node has an edge down to each of its children along with an edge back up to its parent. A node with c children therefore needs $c+1$ elements in the Euler tour array.

Since the Cartesian tree array is constructed such that a child knows its parent but the parent does not know its child, we need a way to deterministically construct the edges from the parent down to its child parallel. Since each node has at most two children, we need to handle three cases for how to write the edges:

- **Zero children:** The thread assigned to the node itself writes to its first element the index of its parent.

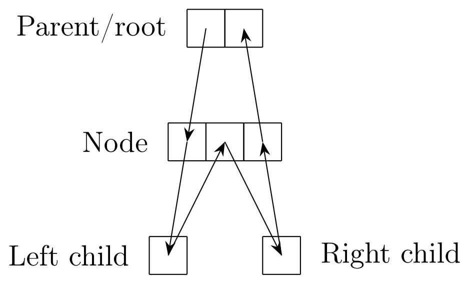

Figure 10: Illustration of the number of the Euler tour of a Cartesian tree with four nodes. This shows how many elements (squares) are assigned to each node based on the number of children, and how these elements are connected by indices (arrows) pointing to the next element to form an Euler tour.

- **One child:** The thread assigned to the child writes to the node's first index the index of the child.
  The thread assigned to the node itself writes to its second element the index of its parent.
- **Two children:** The thread assigned to the left child writes to the node's first index the index of the child. It knows that it is the left child since the index of the child is less that the index of the node.
  The thread assigned to the right child writes to the node's second index the index of the child. It knows that it is the right child since the index of the child is greater that the index of the node.
  The thread assigned to the node itself writes to its third element the index of its parent.

There is an additional set of rules for when we mention "the index of its parent" depending on how many children the parent has.

- **One child:** "The index of its parent" is the parent's second index.
- **Two children:** "The index of its parent" is the parent's third index.

If we apply these rules, we achieve an Euler tour such as e.g. the one shown in fig. 10, where the parent has one child, the "node" has two children denoted left and right child, and the "children" each have no children of their own.

Now that we have a method to construct an Euler tour from a Cartesian tree, we just need to show how to get the relevant information needed. That is the number of children of each node, and the index in the Euler tour array at which the edges related to a specific node is written.

The input to the algorithm is the Cartesian tree and the prefix sum of an array containing the number of edges going out of each node (henceforth $prefixsumedges$). Node i has its designated elements in the Euler tour starting from index $prefixsumedges[i]$. The number of children that node i has can be calculated as $prefixsumedges[i] - prefixsumedges[i-1]$ if $i > 0$, and as $prefixsumedges[i]$ if $i = 0$.

For the CPU to be able to tell the GPU to initialise the Euler tour array, we need to know the length of the array. To avoid having to transfer data from the GPU to the CPU, we note that the number of elements (edges) in the Euler tour array must be twice the number of elements in the Cartesian tree minus one. The reason we know this is that since the Euler tour is based on a tree, each node has one edge from its parent and one edge back to its parent. This accounts for all edges in the Euler tour. The only exception is the root node, which does not have a parent, but still has one edge pointing to outside the array, signalling that it is the end of the path.

While the logic behind the method might be complex, the work needed to be done is a constant amount of calculations and comparisons and it is independent between threads. We therefore implement it with the threads using their global index.

This results in the work performed by a workgroup being $O(n)$ and the span being $O(1)$.

10.7 Kernel 7: Find starting edge in Euler tour

This kernel has the job of finding the index of the first edge in the Euler tour. The input is the Cartesian tree and $prefixsumedges$.

All we do in this kernel is have each thread (using global thread indexes) search for the edge that points outside the array, which must be the root node in the Cartesian tree. Given that the root node has index i, the starting edge of the Euler tour must have index $prefixsumedges[i-1]$ if $i > 0$ and index 0 if $i = 0$.

This kernel obviously performs $O(n)$ work and has $O(1)$ span.

10.8 Kernel 8: Initialise weights W_1 and W_2

Kernel 8 initialises an array with the weights we need for list ranking to calculate preorder, level, and inlabel. For that we need to know which edges go away from the root and which go towards the root.

The input to the kernel is $prefixsumedges$ along with the two values that should be assigned to the edges going away from and towards the root node.

Thread t (global index) finds that the index of node t is $idx = prefixsumedges[t-1]$ if $t > 0$ and $idx = 0$ if $t = 0$, and that the number of edges that node t has is $|edges| = prefixsumedges[i] - prefixsumedges[i-1]$ if $i > 0$ and $|edges| = prefixsumedges[i]$ if $i = 0$. All the edges except the last one go down to the children, so the thread writes the down-value to elements idx to $idx + |edges| - 2$ in the output array, and the up value to element $idx + |edges| - 1$.

Since the number of edges out of each node is at most three, each thread performs constant work. The work is therefore $O(n)$ and the span is $O(1)$.

10.9 Kernel 9: Splitting the linked list

Kernel 9 handles splitting the linked list for the list ranking algorithm into parts.

The input is the index of the start-element of the linked list.

Each thread uses its global index to determine which part of the array containing the linked list that it is responsible for. It then selects a random number in that range, which is the start-element of the part. That number is then written to an output array.

The only exception is when the thread's part contains the start-element for the entire linked list, then it specifically does not pick that element. Thread 0 then ignores its own part, and picks the start-element of the linked list. This ensures that the entire linked list is going to be processed by the list ranking algorithm, and it makes it easier to know the index of the start element for any recursive calls to the linked list algorithm.

This kernel performs $O(n)$ work and has $O(1)$ span.

10.10 Kernel 10: List ranking - calculating part weights

Kernel 10 handles the first half of the list ranking algorithm. Its job is to run through each part of the linked list, calculate its total weight and create a new linked list with each part contracted into one element.

The input is the linked list, weights for the edges, and the indices of the start-elements of each part.

Thread t (global index) starts at start-element t of the linked list and calculates the sum of the weights along the linked list until it reaches another start-element. It then writes the total weight of the part to the output array, along with the index of the other start-element that it found at the end of the part.

This kernel performs $O(n)$ work, and the span is the length of the longest part.

10.11 Kernel 11: List ranking - calculating element ranks

Kernel 11 handles the second half of the list ranking algorithm. Its job is to take the global rank of the start-element of the part, and calculate the global ranks of all elements in the part.

The input is the linked list, weights for the edges, ranks of the start-elements of the parts, and the indices of the start-elements of each part.

Thread t (global index) starts at start-element t and writes its rank to the output. It then follows the linked list and calculates the ranks of the elements in the part by adding their weights to the rank of the start-element.

This kernel performs $O(n)$ work, and the span is the length of the longest part.

10.12 Kernel 12: Calculate preorder, level, and inlabel

The input to this kernel is $prefixsumedges$, D_1, and D_2, which it uses to calculate preorder, level, and inlabel using the formulas described in section 9.1.

This kernel performs $O(n)$ and has $O(1)$ span, since the calculations are independent between the nodes and it performs a constant number of calculations per node.

10.13 Kernel 13: Initialise weights for calculating ascendant

Kernel 13 initialises an array with the weights W_3.

The input is the Cartesian tree, the Euler tour, $prefixsumedges$, and the inlabel array.

It implements the weights as describes by the logic in section 9.1. Since the work is independent between threads, we use the global thread index. Thread t uses the Cartesian tree to find the parent p of node t. It then compares the inlabels of nodes t and p to know what values should be set as the weights of the edges from p to t and back. The only thing left is to find index the edge from p to t in the Euler tour.

Due to the way we constructed the Euler tour, we have obtained the property that end index of the edge from a node to its parent is exactly one larger than the index of the parent to the node. This means that to find the index of the edge from the parent to the node, we use the formula $eulertour[prefixsumedges[t] - 1] - 1$.

The only exception is if t is the root node, but we detect that when we look for the parent using the Cartesian tree. In this case we just do nothing since there are no edges that need weights.

Since we perform a constant number of operations per thread, the work is $O(n)$ and the span is $O(1)$.

10.14 Kernel 14: Calculate ascendant

Kernel 14 calculates the ascendant values of the nodes.

It needs three input values: $prefixsumedges$, D_3, and 2^l which the CPU has precomputed using its knowledge about the size of the Cartesian tree.

Thread t (global index) calculates $ascendant[t] = D_3[prefixsumedges[t-1]] + 2^l$ if $t > 0$ or $ascendant[0] = D_3[0] + 2^l$ if $t = 0$.

This kernel obviously performs $O(n)$ work and its span is $O(1)$.

10.15 Kernel 15: Calculate head

Kernel 15 fills the head table.

Its input is the Cartesian tree and the inlabel array.

Thread t (global index) checks if node t is not the root node, and if its parent has a different inlabel that itself. In this case it writes t at index $inlabel[t]$ in the head table.

This kernel obviously performs $O(n)$ work and its span is $O(1)$.

10.16 Running the entire algorithm

Running the RMQ algorithm is a little more complex than running the ANSV algorithm, since it has more kernels and multiple parts are run recursively. The algorithm is run as follows:

1. Dispatch kernel 1 with $\lceil \frac{n}{32} \rceil$ workgroups for n total threads.
   `cartesian_tree = kernel1(ansv)`

2. Dispatch kernel 2 with $\lceil \frac{n}{32} \rceil$ workgroups for n total threads.
   `childflags = kernel2(cartesian_tree)`

3. Dispatch kernel 3 with $\lceil \frac{n}{32} \rceil$ workgroups for n total threads.
   `num_children = kernel3(childflags)`

4. Dispatch kernel 4 with $\lceil \frac{n}{32} \rceil$ workgroups for n total threads.
   We initialise an array `allones` with $\lceil \frac{n}{32} \rceil$ ones.
   `num_edges = kernel4(num_children, allones)`

5. Kernel 5 calculates the workgroup-local prefix sum. To calculate the global prefix sum, we use the method described in section 9.3.1. We only show it with one level of recursion, but it should continue until one workgroup handles the entire prefix sum at the innermost layer of the recursion.
   For one level of recursion where each workgroup handles m_1 elements, we:

   - Dispatch kernel 5 with $\left\lceil \frac{n}{m_1} \right\rceil$ workgroups, with $m < n \leq m_1^2$.
   - Dispatch kernel 5 with one workgroup.
   - Dispatch kernel 4 with $\left\lceil \frac{n}{m_1} \right\rceil$ workgroups.

   ```
   prefix_sum_num_edges_1 = kernel5(num_edges)
   prefix_sum_num_edges_2 = kernel5(prefix_sum_num_edges_1)
   prefix_sum_num_edges = kernel4(prefix_sum_num_edges_1, prefix_sum_num_edges_2)
   ```

6. Dispatch kernel 6 with $\lceil \frac{n}{32} \rceil$ workgroups for n total threads.
   `euler_tour = kernel6(cartesian_tree, prefix_sum_num_edges)`

7. Dispatch kernel 7 with $\lceil \frac{n}{32} \rceil$ workgroups for n total threads.
   `euler_tour_root = kernel7(cartesian_tree, prefix_sum_num_edges)`

8. Dispatch kernel 8 with $\lceil \frac{n}{32} \rceil$ workgroups for n total threads.
   ```
   W1 = kernel8(cartesian_tree, prefix_sum_num_edges, 0, 1)
   W2 = kernel8(cartesian_tree, prefix_sum_num_edges, -1, 1)
   ```

9. Dispatch kernel 9 with $\left\lceil \frac{n}{m_2} \right\rceil$ workgroups for $\left\lceil \frac{n}{m_2} \right\rceil$ total threads.
   The threads implicitly knows m_2, so they can calculate indices of the part that it splits without needing further input.
   `splits = kernel9(euler_tour_root)`

10. Kernel 10 and 11 performs the list ranking algorithm described in section 9.3.2. We only show it with one level of recursion, but the recursion should continue until one workgroup calculates the ranks for the entire list.
    For one level of recursion where each workgroup handles m_2 elements, we:
    - Dispatch kernel 10 with $\left\lceil \frac{n}{m_2} \right\rceil$ workgroups, with $m < n \leq m_2^2$.
    - Dispatch kernel 10 with one workgroup.
    - Dispatch kernel 11 with $\left\lceil \frac{n}{m_2} \right\rceil$ workgroups.

    We initialise an array `allzeroes` with $\left\lceil \frac{n}{m_2} \right\rceil$ zeroes.
    ```
    part_linked_list, part_weights = kernel10(splits, W1, euler_tour,
                                              euler_tour_root)
    ```
    The CPU calculates the start-element/root of `part_linked_list` from m and `euler_tour_root`.
    ```
    part_splits = kernel9(part_linked_list_root)
    part_ranks = kernel11(part_splits, part_weights, part_linked_list, allzeroes)
    D1 = kernel11(splits, W1, linked_list, part_ranks)
    ```
    All of this is then run twice. Once with the weights `W1`, and once with `W2` in order to calculate `D2`.

11. Dispatch kernel 12 with $\lceil \frac{n}{32} \rceil$ workgroups for n total threads.
    `preorder, level, inlabel = kernel12(prefix_sum_num_edges, D1, D2)`

12. Dispatch kernel 13 with $\lceil \frac{n}{32} \rceil$ workgroups for n total threads.
    `W3 = kernel13(cartesian_tree, euler_tour, prefix_sum_num_edges, inlabel)`

13. Repeat step 10. but with the weights `W3` in order to calculate `D3`.

14. Dispatch kernel 14 with $\lceil \frac{n}{32} \rceil$ workgroups for n total threads.
    The CPU calculates 2^l using its knowledge of n.
    `ascendant = kernel14(prefix_sum_num_edges, D3, 2^l)`

15. Dispatch kernel 15 with $\lceil \frac{n}{32} \rceil$ workgroups for n total threads.
    `head = kernel15(cartesian_tree, inlabel)`

Despite us describing that for many of the kernels each thread thread performs the calculations related to one node in the Cartesian tree, in practice we actually have each thread handle a constant number of calculations in order to reduce the overhead of spawning a thread.

10.17 Correctness

The method we use to answer LCA queries and the list ranking algorithm are both described and proven to be correct in their respective articles. Instead, we need to show that our many steps of running small kernels actually run those algorithms.

The necessary argument for each kernel are already given in the description of the kernels, so here we will describe what the kernels do when put together.

Kernel 1 converts the ANSV result to a Cartesian tree as described in (Berkman et al., 1993).

Kernels 2 and 3 counts the number of children of each node as we showed in the description of kernel 2.

Kernel 4 then adds one to the number of children to get the number of edges out of a node, according to the description of kernel 6.

Kernel 5 is then used together with kernel 4 to calculate the prefix sum of the number of edges out of each node. This data can then be used in a kernel to calculate both the number of edges out from a node along with those edges' along with unique indices for each of those edges.

Kernel 6 then constructs an Euler tour using the prefix sum of the number of edges out of each node.

So far kernel 1 through 6 has been used to calculate the Cartesian tree and an Euler tour through said tree. (Schieber & Vishkin, 1988) then describes that to answer LCA queries, we need to set up specific weights for the edges in the Euler tour and run a list ranking algorithm using those weights.

Kernel 8 sets the first two sets of weights, and kernels 9, 10, and 11 then runs the recursive Helman-JáJá algorithm as described in (Rehman et al., 2009) in order to solve the list ranking problem.

Kernel 12 can then calculate the preorder, level, and inlabel arrays as described in (Schieber & Vishkin, 1988).

Kernel 13 can then use the inlabel array to set up the third set of weights, after which we once more run the list ranking algorithm.

Kernel 14 and 15 then calculates the ascendant array and the head table as described in (Schieber & Vishkin, 1988).

This shows that while the algorithm has been split into many small parts in order to fit with the architecture of a GPU, it is still the same algorithm and it is therefore still correct.

10.18 Work and span

Before we analyse the algorithm, we would like to point out that almost the entire algorithm can run on $\frac{n}{\log(n)}$ threads performing $O(n)$ work with $O(\log^2(n))$ span, just like the ANSV algorithm. There is an exception, however, which is when we run the list ranking algorithm. In (Helman & JáJá, 2001) Helman and JáJá analysed the list ranking algorithm and their analysis only holds when the total number of threads, p, satisfies $n > p^2 \ln(n)$.[8] That is if $p < \sqrt{\frac{n}{\ln(n)}}$.

We will therefore limit the analysis to using $\sqrt{\frac{n}{\ln(n)}} - 1$ threads, despite most parts supporting further parallelism.

Furthermore, the analysis assumes that the array is split into s parts, with each thread handling $\frac{s}{p}$ parts. s needs to be $\geq p\ln(n) + 1$. So with $p = \sqrt{\frac{n}{\ln(n)}} - 1$ we the lower bound on s is

$$s \geq p\ln(n) + 1$$

[8]Helman and JáJá, 2001, Lemma 1, p. 270.

$$= \left(\sqrt{\frac{n}{\ln(n)}} - 1\right)\ln(n) + 1$$
$$= \sqrt{n\ln(n)} - \ln(n) + 1.$$

We decide on $s = \sqrt{n\ln(n)}$ for our analysis of the list ranking algorithm.

Work

Kernels 1, 2, 3, 6, 7, 8, 12, 13, 14, and 15 are all only run once on an input of length $O(n)$. Since the amount of work performed per input element is $O(1)$, the total work is $O(n)$. The remaining kernels are therefore 4, 5, 9, 10, and 11.

Kernel 4 is first run once to convert `num_children` into `num_edges`, during which it performs $O(n)$ work.

Kernel 4 and 5 are then run recursively. Given an input size of n and that we assign m elements to each workgroup, each iteration of the recursion will reduce the input size of the next iteration to $\left\lceil \frac{n}{m} \right\rceil$. Since we dispatch $\left\lceil \frac{n}{m} \right\rceil$ workgroups per iteration, and each workgroup performs $O(m)$ work in each of the two kernels, we end up with the recurrence relation $T(n) = T\left(\frac{n}{m}\right) + cn$ describing the total work performed during the recursion.

If we assume that we have $p = \sqrt{\frac{n}{\ln(n)}} - 1$ threads, and split n evenly between workgroups, we get $m = \frac{n}{\left(\sqrt{\frac{n}{\ln(n)}}-1\right)\cdot 32}$ elements per workgroup. Just like in section 7.5.9, we guessing that the work is $T(n) \leq 2cn$ and plugging it into the recurrence.

Base case $(n = 137)$

$$\begin{aligned} T(137) &= c && \text{by definition} \\ 2cn = 4c \geq c &= T(137) = T(n). \end{aligned}$$

Induction step

Assume $T(\ell) \leq 2c\ell$ for $137 \leq \ell < n$.

$$\begin{aligned} T(n) &= T\left(\frac{n}{m}\right) + cn \\ &= T\left(\frac{n}{\frac{n}{\left(\sqrt{\frac{n}{\ln(n)}}-1\right)\cdot 32}}\right) + cn \\ &\leq 2c\frac{n}{\frac{n}{\left(\sqrt{\frac{n}{\ln(n)}}-1\right)\cdot 32}} + cn \\ &= 2c\left(\sqrt{\frac{n}{\ln(n)}} - 1\right)\cdot 32 + cn \\ &= c\left(64\frac{\sqrt{n}}{\sqrt{\ln(n)}} - 64\right) + cn \\ &\leq 2cn = O(n) \end{aligned}$$

We note that we assumed that $n \geq 137$ in order to make the inequality $64\frac{\sqrt{n}}{\sqrt{\ln(n)}} - 64 \leq 2n$ hold.

The other kernels are 9, 10, and 11, which execute the recursive Helman-JáJá list ranking algorithm. When we run kernel 9, we split the array into parts, and pick one start-element from each part. As we mentioned above, we decided to set the number of parts to $s = \sqrt{n \ln(n)}$. All three kernels perform $O(n)$ work per iteration, so we need solve the recurrence relation $T(n) = T(s) + cn$. We guess the solution to be $T(n) \leq 2cn$ and try to solve it.

Base case $(n = 9)$

$$\begin{aligned} T(9) &= c && \text{by definition} \\ 2cn &= 4c \geq c = T(9) = T(n). \end{aligned}$$

Induction step
Assume $T(\ell) \leq 2c\ell$ for $9 \leq \ell < n$.

$$\begin{aligned} T(n) &= T(s) + cn \\ &\leq 2cs + cn \\ &= 2c\sqrt{n \ln(n)} + cn \\ &\leq 2c\frac{n}{2} + cn \\ &= 2cn \end{aligned}$$

We note that we assumed that $n \geq 9$ in order to make the inequality $\sqrt{n \ln(n)} \leq \frac{n}{2}$ hold.

To summarise, the work performed by the entire algorithm is $O(n)$, meaning that it is work optimal.

Span
Kernels 1, 2, 3, 6, 7, 8, 12, 13, 14, and 15 are all only run once on an input of size $O(n)$, and all have $O(1)$ span assuming that we assigned one thread to each element. This means that since we have $\sqrt{\frac{n}{\ln(n)}} - 1$ threads, if we distribute the elements equally between threads, for $n \geq 9$ each thread would handle $\frac{n}{\sqrt{\frac{n}{\ln(n)}}-1}$ elements, resulting in the span being

$$\begin{aligned} \frac{n}{\sqrt{\frac{n}{\ln(n)}} - 1} &\leq \frac{2n}{\sqrt{\frac{n}{\ln(n)}}} \\ &= \frac{2n}{\frac{\sqrt{n}}{\sqrt{\ln(n)}}} \\ &= 2\sqrt{n \ln(n)} \\ &= O\left(\sqrt{n \log(n)}\right). \end{aligned}$$

The remaining kernels are once again 4, 5, 9, 10, and 11.

Kernel 4 is first run once with $O(1)$ span on n threads, or $O\left(\sqrt{n \log(n)}\right)$ span on p threads, just as the above mentioned kernels.

The recursion of kernel 4 and 5 performs $O(m)$ work per workgroup, and since each iteration reduces the input size to $\lceil \frac{n}{m} \rceil$ we get the recurrence $T(n) = T\left(\frac{n}{m}\right) + cm$. If we split the input array equally, then $m = \frac{n}{\left(\sqrt{\frac{n}{\ln(n)}}-1\right)\cdot 32}$ and if we guess that $T(n) \leq 64c\sqrt{n\ln(n)}$, we can solve the recurrence.

Base case $(n = 1905)$

$$\begin{aligned} T(1905) &= c && \text{by definition} \\ 64c\sqrt{n\ln(n)} = 64c\sqrt{1905\ln(1905)} &\geq c = T(1905) = T(n) \end{aligned}$$

Induction step
Assume $T(\ell) \leq 64c\sqrt{\ell\ln(\ell)}$ for $1905 \leq \ell < n$.

$$\begin{aligned} T(n) &= T\left(\frac{n}{m}\right) + cm \\ &= T\left(\frac{n}{\frac{n}{\left(\sqrt{\frac{n}{\ln(n)}}-1\right)\cdot 32}}\right) + cm \\ &= T\left(\left(\sqrt{\frac{n}{\ln(n)}} - 1\right)\cdot 32\right) + c\left(\sqrt{\frac{n}{\ln(n)}} - 1\right)\cdot 32 \\ &\leq 64c\sqrt{\left(\sqrt{\frac{n}{\ln(n)}} - 1\right)\cdot 32\ln\left(\left(\sqrt{\frac{n}{\ln(n)}} - 1\right)\cdot 32\right)} + c\left(\sqrt{\frac{n}{\ln(n)}} - 1\right)\cdot 32 \\ &\leq 64c\sqrt{\frac{n}{4}\ln(n)} + 32c\sqrt{n\ln(n)} \\ &= 64c\sqrt{n\ln(n)} = O(\sqrt{n\log(n)}) \end{aligned}$$

We note that we assumed that $n \geq 1905$ in order to make the inequality $\left(\sqrt{\frac{n}{\ln(n)}} - 1\right)\cdot 32 \leq \frac{n}{4}$ hold, which at the same time also gives us the inequality $\sqrt{\frac{n}{\ln(n)}} - 1 \leq \sqrt{n\ln(n)}$.

For the analysis of the span of the recursion performed by kernels 9, 10, and 11, we use the analysis of Helman and JáJá which states that when we split the linked list into s parts, with each of the p threads handling $\frac{s}{p}$ parts, then each thread handles at most $\alpha(s)\frac{n}{p}$ elements with high probability (meaning with probability greater than $1-n^{-\epsilon}$ for some $\epsilon > 0$), for some function α of s, $\alpha(s) \geq 2.62$.[9] While $\alpha(s)$ does not look like a constant, Helman and JáJá use the fact that each thread handles $\alpha\frac{n}{p}$ elements to show that the computation time of the threads to traverse their respective parts of the linked list is $O\left(\frac{n}{p}\right)$. We therefore assume that $\alpha(s)$ is a constant.

Explained in simpler terms, this all means that each thread handles approximately the same number of elements.

Since we split the array into s parts, the span of the recursive list ranking algorithm can be explained by the recurrence relation $T(n) = T(s) + c\alpha\frac{n}{p}$. If we guess that $T(n) = 2\alpha c\sqrt{n\ln(n)}$ we can solve the recurrence.

[9]Helman and JáJá, 2001, Lemma 1, p. 270.

Base case $(n = 68)$

$$\begin{aligned} T(68) &= c && \text{by definition} \\ 2\alpha c\sqrt{n\ln(n)} = 2\alpha c\sqrt{68\ln(68)} &\geq c = T(68) = T(n) \end{aligned}$$

Induction step
Assume $T(\ell) \leq 2\alpha c\sqrt{\ell\ln(\ell)}$ for $68 \leq \ell < n$.

$$\begin{aligned} T(n) &= T(s) + c\alpha\frac{n}{p} \\ &\leq 2\alpha c\sqrt{s\ln(s)} + c\alpha\frac{n}{p} \\ &= 2\alpha c\sqrt{\sqrt{n\ln(n)}\ln(\sqrt{n\ln(n)})} + c\alpha\frac{n}{\sqrt{\frac{n}{\ln(n)}} - 1} \\ &= 2\alpha c\sqrt[4]{n\ln(n)}\sqrt{\ln(\sqrt{n\ln(n)})} + c\alpha\frac{n}{\sqrt{\frac{n}{\ln(n)}} - 1} \\ &\leq 2\alpha c\sqrt[4]{n\ln(n)}\sqrt{\ln(n)} + c\alpha\frac{n}{\sqrt{n\ln(n)}} \\ &\leq 2\alpha c\frac{\sqrt{n}}{2}\sqrt{\ln(n)} + c\sqrt{n}\frac{\alpha}{\sqrt{\ln(n)}} \\ &\leq 2\alpha c\sqrt{n\ln(n)} = O(\sqrt{n\log(n)}) \end{aligned}$$

We note that we assumed that $n \geq 1905$ in order to make the inequality $\sqrt[4]{n\ln(n)} \leq \frac{\sqrt{n}}{2}$ hold, which at the same times also gives us the inequality $\sqrt{\frac{n}{\ln(n)}} - 1 \leq \sqrt{n\ln(n)}$.

Using $p = \sqrt{\frac{n}{\ln(n)}} - 1$ threads, the total span of the algorithm becomes $O(\sqrt{n\log(n)})$.

In (Rehman et al., 2009) it is claimed that if we let $p = \frac{n}{\log(n)}$ and split the linked list into in p parts, the expected span can be expressed by the recurrence relation $T(n) = T(p) + O\left(\frac{n}{p}\right)$. Using this, we would find the span of the recursive Helman-JáJá algorithm to be $O(\log^2(n))$, which means the entire algorithm would perform $O(n)$ work and have $O(\log^2(n))$ span on $\frac{n}{\lg(n)}$ threads.

In the article they do not, however, explain how they handle the fact that the lengths of the parts of the linked list are dependent on each other. If one part is shorter than the average length, then those missing elements must be present in other parts, making those longer than the average length. Since they do not explain how they handle this issue, or even mention its existence, we decided to not rely on this result.

11 Generalised RMQ

While the explanation of the algorithm itself is done, we have one final addition. If we use the ANSV of an array with duplicated value as input to the RMQ algorithm, we do not necessarily find the correct output. The reason is that we might end up with the LCA of two nodes a and b being outside the interval $[a; b]$. An example of this case can be seen in fig. 11, where the black and blue edges form the Cartesian tree, so it can easily be seen that the LCA of a and b is p, which is left of both of them.

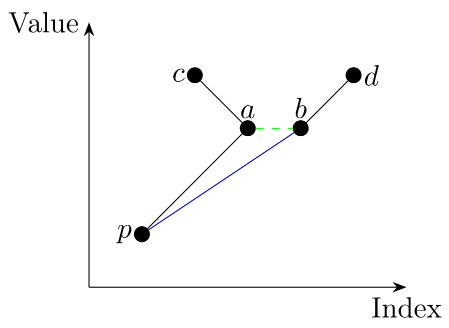

Figure 11: The black and blue edges form the tree found when trying to construct a Cartesian tree by running an ANSV algorithm. If we replace the blue edge with the green edge, we find a tree capable of answering RMQ queries.

This issue only occurs when two nodes a and b, share their value, share a parent p, and they are either both left or both right of the parent. In this case, when we want to answer the RMQ query for a descendant of a and a descendant of b (or a and b themselves), the answer will always be the value of a (or b since they are the same value), but the LCA that we find is p. To make the LCA become a, we remove the edge from b to p, and instead insert an edge from b to a, letting a be the parent of b. In fig. 11, this is when we replace the blue edge with the green edge.

Luckily, constructing such a Cartesian tree is easy. As we mentioned in section 5, (Berkman et al., 1993)'s suggested method of solving the generalised ANSV problem is by enumerating all the elements and using the index as a tie-breaker for whenever two element share the same value. Since this would cause neighbouring elements which share the same value to find each other as the nearest smaller value, but only in the direction of the lowest index, the article suggested running the algorithm twice, once where the indices are increasing and once where they are decreasing. This issue, however, is perfect for solving the generalised RMQ.

If we enumerate the elements, and only run the ANSV algorithm once, neighbouring elements with the same value will form a directed path, and only one of them will have their parent be the element that is the actual nearest smaller value of all the elements. Explained in another way, we would get the tree from fig. 11 but with the green edge instead of the blue edge, just as we wanted. It is therefore easy to solve the generalised RMQ using the algorithms that we have described so far.

11.1 Implementation and test setup

For our benchmarks we only have two implementations:

1. Our GPU algorithm.
2. The `range_minimum_query` Rust crate (Rust terminology for package).[10]

   We would have liked to implement the RMQ algorithm ourselves, but we ran out of time. This serves as a substitution, since it claims to utilise a Cartesian tree to answer RMQ queries in constant time. Sadly, it is not a parallelised algorithm.

When implementing our GPU algorithm, we had trouble implementing kernel 9. Specifically, since RustGPU is still early in development, we couldn't find a way to generate the random numbers

[10]Petri, 2024.

needed for kernel 9. Generating random number is a common feature available when using other GPU code compilers, so it is not an issue with the algorithm itself. It is only with our implementation.

What we decided to do instead is to "randomly" pick the first element of each part to be the start-element of the part, with the second element being the backup start-element in case the global start-element of the linked list is the first element in some part.[11]

11.2 Test setup

For our RMQ benchmarks we will be utilising the same datasets as we did for the ANSV benchmarks, but with the number of elements decreased from 500 million to 50 million in order to accommodate the larger memory usage of the RMQ algorithm.

For the GPU algorithm, since its input is the output of the ANSV algorithm, we first run the GPU ANSV algorithm on the dataset, and then we run the GPU RMQ algorithm.

The input to the CPU algorithm is just an array of data, so we just input the dataset directly.

While the datasets might seem randomly chosen and no longer hand picked, they each still serve two distinct functions.

- Since we couldn't randomly pick the start-elements in kernel 9, we could reintroduce the randomness by letting the data itself be random. While we can't control how running the ANSV algorithm followed by constructing a Cartesian tree and an Euler tour affects the randomness of the input, we can at make it as random as possible by letting the input be random data, i.e. the random dataset.

- The worst case dataset for the ANSV algorithm will actually be close to a best case dataset for the RMQ algorithm. Due to the way the worst case dataset is constructed, it is actually a bitonic array. The Cartesian tree of a bitonic array forms a single path, with the root element being either the leftmost or rightmost element, and then the next element in the path is either the rightmost or the leftmost element that is not already in the path.

  When we then construct an Euler tour, all nodes in the Cartesian tree except the leaf node will have two outgoing edges in the Euler tour, while the leaf node will have one outgoing edge. Due to the exact way we construct the Euler tour (see section 10.6), each on the path from the root to the leaf node, all even indices of the Euler tour in the range $\left[0; \frac{n}{2}\right]$ and all odd indices in the range $\left[\frac{n}{2}; n\right]$ will be traversed, with the indices from each range being traversed in sorted order. On the path back from the leaf to the root, it is the odd indices in the range $\left[0; \frac{n}{2}\right]$ and the even indices in the range $\left[\frac{n}{2}; n\right]$ that are traversed.

  Now, since we place the start-elements at regular intervals through the array, the maximum distance between two start-elements will be twice the average. The reason is that if the space between the start-elements is an even number of elements, then we will encounter all start-elements in the range $\left[0; \frac{n}{2}\right]$ on the path from the root to the leaf node, and then we encounter all the start elements in the range $\left[\frac{n}{2}; n\right]$ on the path back to the root. If the space between the start-elements is an odd number, then we encounter half of the start-elements from each range on the path to the leaf node, and the other half on the path back to the root.

The hardware that we utilise is the AMD Ryzen 5 7600x CPU (since it is the fastest single-threaded CPU we have available), and all the GPUs that we used for the ANSV benchmarks.

[11] https://xkcd.com/221/

11.3 Results

The data that we have collected can be seen in table 8. We first notice that the RADV driver is no longer much faster than the AMDGPU Pro driver. The relative performance difference between the GPUs, however, is still the same as for the ANSV benchmarks, which tells is that the AMDGPU Pro driver was just particularly bad at something that we did for the implementation of the ANSV algorithm.

Since the purpose of the RMQ algorithm was to utilise the ANSV output to easily solve another problem, this GPU implementation is a bit of a failure. Compared to running the GPU ANSV algorithm, it takes about 7 to 11 times longer to process the ANSV output for answering RMQ queries. As we will mention in the optimisations section, there are a few optimisations that we didn't have time to implement, but overall, it might just be better to design an algorithm that solves the RMQ problem from scratch, rather than trying to utilise the already computed ANSV output.

Additionally, if we compare it to the CPU version, the GPU algorithm is only 3 to 6 times faster. And since the CPU version is sequential, it should be relatively easy to make it faster than the GPU algorithm by parallelising it with the algorithm that we based our GPU algorithm on.

Our results reveal one good thing though. We analysed the algorithm using the assumption that $n > p^2 \ln(n)$. The AMD Radeon RX 7900XTX GPU has 96 compute units, each with 4 SIMDs, which handles workgroups with 32 threads. This means that in total, we have 12288 threads. This means that the inequality if only satisfied when $n \approx 3{,}310{,}000{,}000$, or 6.6 times larger than the maximum buffer size the GPU supports (not including the fact that the Euler tour buffer is up $3n$ long). Yet somehow, the data showed no significant difference in performance between the random dataset and the worst case dataset (which is actually the best case dataset) where we do not rely on the randomness analysis. This means that in practice, the recursive Helman-JáJá list ranking algorithm should scale much better than what we found theoretically.

11.4 Optimisations

When we tried optimising the RMQ GPU algorithm, we once again used the RadeonGPUProfiler. As seen in fig. 12, it is very obvious that if this algorithm is become competitive with the CPU algorithm, it must optimise the time it takes to perform list ranking.

The we started by optimising the size of the parts that the linked list are split into, and found that 300 elements was close to optimal, but with a big exception. Since it is slow to dispatch GPU kernels, we found that it was worth it to end the recursion when the length of the part-sums array became 15,000 elements, and then have just a single thread compute the ranks sequentially. It might be worth testing if it is faster to copy the last part-sums array to the CPU and have it compute the ranks instead. We sadly didn't have time to test this.

Another easy optimisation is that the first two times we run the list ranking algorithm, first using weights W_1 and then using W_2, it might be possible to combine these into one. The reason why this might be efficient, is that the main bottleneck for the list ranking algorithm is the memory access latency, and not the computations themselves. If we therefore merge W_1 and W_2 into one array with it the output array alternating between elements from W_1 and elements from W_2, we might achieve the exact same memory access latency, which halving the number of memory accesses, therefore running the list ranking algorithm twice in the time it takes to run it once.

Sadly, the third instance of the list ranking algorithm depends on the output of the previous two, so it cannot be combined with the other two.

Even if we implement all these optimisations we are unlikely to get much more than a 20%-30% (based on the timeline in fig. 12 which is obtained while collecting profiling information, so it is very

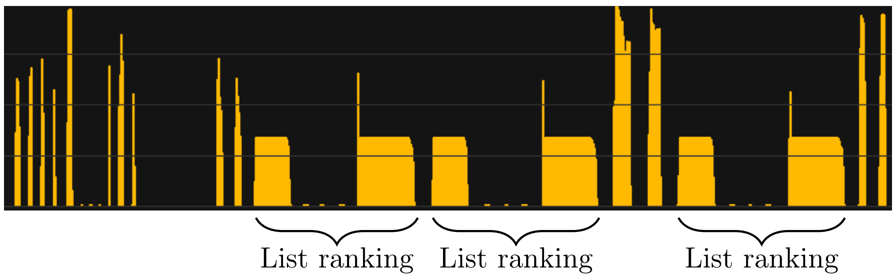

Figure 12: A screenshot of part of the output when profiling the GPU RMQ algorithm using the RadeonGPUProfiler software. The horizontal axis represents the time spent, and the vertical axis represents how "busy" the GPU is measured as the number of workgroups actively running on the hardware. The marked parts are where kernels 9, 10, and 11 are dispatched to run a list ranking algorithm.

inaccurate) or maybe a 50% speed-up if we are very lucky. While this is a large speed-up, if we look at the data in table 8, we see that the RMQ algorithm would still be much slower than th ANSV algorithm. And it also wouldn't be much faster than a hypothetical parallelised CPU algorithm.

Our best guess of why the GPU algorithm performs so poorly, is because of memory access latency. What makes GPUs fast is that they can perform many calculations in parallel and have the SIMDs swap between workgroups in order to hide memory access latency. The list ranking algorithm is the antithesis of this. It performs a random memory access to get a single element, after which it adds two numbers and performs another random memory access. This means that the SIMD cannot keep progressing workgroups, as all the workgroups in the cache are waiting for memory accesses to finish.

The best way to remedy this is to have the entire linked list in the cache, but a GPU has much less cache space per thread than a CPU, so when the input data is large enough that a GPU becomes relevant, it will have no chance of keeping the linked list in the cache.

Part III
Conclusion

In this thesis we have optimised a CPU ANSV algorithm and designed and implemented a GPU version of both an ANSV algorithm and a RMQ algorithm.

Our optimised CPU ANSV algorithm enables calculating the ANSV of input with duplicated values by running it once. The original algorithm is designed for input with distinct values, and only supports duplicated values by running it twice. Our algorithm is therefore twice as fast on inputs with duplicated values.

Additionally, our changes remove the need for running a prefix- and suffix minima algorithm as part of the ANSV algorithm, which means that in our testing, our algorithm runs faster than the original.

Our algorithm was, however, still slower than the BSZ algorithm implementation found at (Shun & Zhao, 2013a).

We designed and implemented a GPU version of the ANSV algorithm. We found that it is performs $O(n)$ work and has $O(\log^2(n))$ span on $\frac{n}{\lg(n)}$ threads on an CREW PRAM, making it only slightly worse than the $O(n)$ work and $O(\log(n))$ span on $\frac{n}{\lg(n)}$ threads of the original CPU ANSV algorithm that we based it on. In return, our version supports running on a GPU on which we achieved 4-5 times better performance than the CPU algorithm.

We then designed and implemented a GPU version of an algorithm that preprocesses the ANSV of an array in order to answer RMQ queries in constant time. Following the constraints mentioned in (Helman & JáJá, 2001, Lemma 1, p. 270), we analysed our algorithm to perform $O(n)$ work with $O(\sqrt{n\log(n)})$ span on $\sqrt{\frac{n}{\ln(n)}} - 1$ threads on an CREW PRAM. In our tests, however, we found that despite us breaching the constraints by having too many threads, the worst case scenario of one thread traversing a large part of the linked list never happened, so we suspect that the limit on the number of threads might not be relevant in practice, turning it into a $O(n)$ work and $O(\log^2(n))$ span algorithm on $\frac{n}{\lg(n)}$ threads.

12 Future work

The three main issues we couldn't solve were:

- The GPU ANSV algorithm being limited by cache.
- The GPU RMQ algorithm being bottlenecked by the list ranking algorithm.
- Analysing the Helman-JáJá list ranking algorithm in such a way that it no longer has a theoretical upper limit on the number of threads.

To improve on the GPU ANSV algorithm, it would likely require finding a way to replace the kernel that calculates the local ANSV of a subset, and the kernel that runs the merging procedure with other kernels that perform the same work without relying on having large amounts of data in the cache at a time.

Likewise, due to the recursive Helman-JáJá list ranking algorithm relying so heavily on random memory accesses, it would likely require finding another list ranking algorithm in order to improve significantly on our GPU RMQ algorithm.

13 Thanks

Thanks to Professor Inge Li Gørtz and Professor Philip Bille for support and good guidance throughout the project.

Thanks to Albert Sidenius Garde for helping me figure out proc macros in Rust, which made optimising GPU kernels easier.

References

Berkman, O., Schieber, B., & Vishkin, U. (1993). Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, *14*(3), 344–370. https://doi.org/https://doi.org/10.1006/jagm.1993.1018

Helman, D. R., & JáJá, J. (2001). Prefix computations on symmetric multiprocessors. *Journal of Parallel and Distributed Computing*, *61*(2), 265–278. https://doi.org/https://doi.org/10.1006/jpdc.2000.1678

Petri, M. (2024). *Range_minimum_query*. Retrieved June 25, 2025, from https://docs.rs/range_minimum_query/latest/range_minimum_query/

Rehman, M. S., Kothapalli, K., & Narayanan, P. J. (2009). Fast and scalable list ranking on the gpu. *Proceedings of the 23rd International Conference on Supercomputing*, 235–243. https://doi.org/10.1145/1542275.1542311

Schieber, B., & Vishkin, U. (1988). *On finding lowest common ancestors: Simplification and parallelization*. In J. H. Reif (Ed.), *Vlsi algorithms and architectures* (pp. 111–123). Springer New York.

Shun, J., & Zhao, F. (2013a). *Practical parallel lempel-ziv factorization*. Retrieved June 24, 2025, from https://github.com/zfy0701/Parallel-LZ77

Shun, J., & Zhao, F. (2013b). Practical parallel lempel-ziv factorization. *2013 Data Compression Conference*, 123–132. https://doi.org/10.1109/DCC.2013.20

Sitchinava, N., & Svenning, R. (2024). The all nearest smaller values problem revisited in practice, parallel and external memory. *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, 259–268. https://doi.org/10.1145/3626183.3659979

A Data points from the ANSV benchmarks

Hardware\Threads	1	6	12	16	24	32
Ryzen 5 7600x	8314ms	1640ms	1350ms			
Xeon E5-2650 v4	23422ms	4122ms	2383ms	2424ms	2495ms	
Xeon Gold 6226R	22283ms	4044ms	2382ms	2324ms	2422ms	2515ms

Table 1: Benchmarks of the original CPU ANSV algorithm on the random dataset. Red data points designate when we utilise multithreading.

Hardware\Threads	1	6	12	16	24	32
Ryzen 5 7600x	6607ms	1281ms	907ms			
Xeon E5-2650 v4	17670ms	3095ms	1645ms	1654ms	1665ms	
Xeon Gold 6226R	17172ms	3111ms	1588ms	1250ms	1254ms	1267ms

Table 2: Benchmarks of our modified CPU ANSV algorithm on the random dataset. Red data points designate when we utilise multithreading.

Hardware\Threads	1	6	12	16	24	32
Ryzen 5 7600x	6484ms	1242ms	873ms			
Xeon E5-2650 v4	15899ms	2681ms	1363ms	1043ms	727ms	
Xeon Gold 6226R	14621ms	2463ms	1235ms	940ms	646ms	510ms

Table 3: Benchmarks of the BSZ algorithm on the random dataset. Red data points designate when we utilise multithreading.

Hardware\Threads	1	6	12	16	24	32
Ryzen 5 7600x	4809ms	1032ms	1050ms			
Xeon E5-2650 v4	14872ms	2803ms	1794ms	1839ms	1895ms	
Xeon Gold 6226R	14207ms	2760ms	1839ms	2198ms	2326ms	2449ms

Table 4: Benchmarks of the original CPU ANSV algorithm on the worst case dataset. Red data points designate when we utilise multithreading.

Hardware\Threads	1	6	12	16	24	32
Ryzen 5 7600x	4666ms	908ms	871ms			
Xeon E5-2650 v4	13657ms	2414ms	1317ms	1315ms	1328ms	
Xeon Gold 6226R	13237ms	2387ms	1280ms	1021ms	1029ms	1050ms

Table 5: Benchmarks of our modified CPU ANSV algorithm on the worst case dataset. Red data points designate when we utilise multithreading.

Hardware\Threads	1	6	12	16	24	32
Ryzen 5 7600x	3560ms	803ms	687ms			
Xeon E5-2650 v4	11486ms	2276ms	1155ms	898ms	642ms	
Xeon Gold 6226R	11702ms	2205ms	1105ms	843ms	589ms	466ms

Table 6: Benchmarks of the BSZ algorithm on the worst case dataset. Red data points designate when we utilise multithreading.

Hardware\Dataset	random	worst case
AMD Radeon RX 7900XTX (RADV)	98ms	118ms
AMD Radeon RX 7900XTX (AMDGPU Pro)	495ms	382ms
Nvidia Tesla V100	429ms	439ms
Nvidia Tesla A100	169ms	196ms

Table 7: Benchmarks of the GPU ANSV algorithm.

B Data points from the RMQ benchmarks

Hardware\Dataset	random ANSV	RMQ	worst case ANSV	RMQ
AMD Radeon RX 7900XTX (RADV)	12ms	114ms	11ms	125ms
AMD Radeon RX 7900XTX (AMDGPU Pro)	50ms	119ms	35ms	122ms
Nvidia Tesla V100	38ms	429ms	37ms	439ms
Nvidia Tesla A100	24ms	169ms	26ms	196ms
Ryzen 5 7600x (sequential)	716ms		360ms	

Table 8: Benchmarks of the GPU RMQ algorithm, split into the time to run the ANSV algorithm and the RMQ algorithm. The CPU test has the two algorithms combined into one datapoint.