A Parallel Algorithm for Inclusive Scan

1 Project Idea

A sequential method of performing an `inclusive_scan` consists of computing a single sum of two elements in an array in each iteration. For an array of size n, the operation is performed as follows:

```
for(uint8_t index = 1; index < n; index++){
    array[i] = array[i-1] + array[i];
}
```

This cumulative operation is computed in O(n) time complexity.

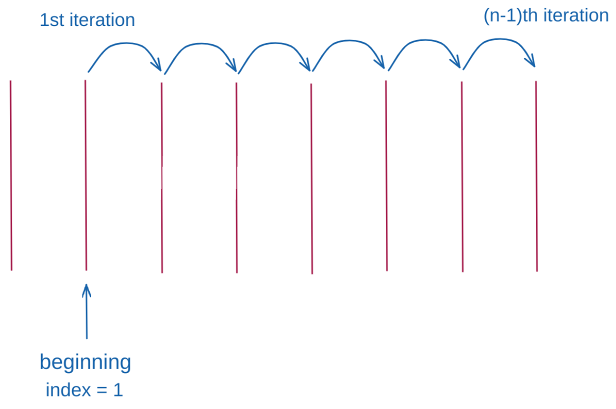

Figure 1.1: *Sequential Inclusive Scan*

In contrast, a parallel version of the aforementioned algorithm could make use of an inverse binary tree to guarantee thread-safety and time efficiency. Specifically, the array operations are divided by 2 at each level and each thread is accountable for a single operation that involves elements of the array that are not part of another thread's operation. This method is known as the `reduce` algorithm. The diagram below is representative of the algorithm's function in case of an array with $n = 2^l$ elements.

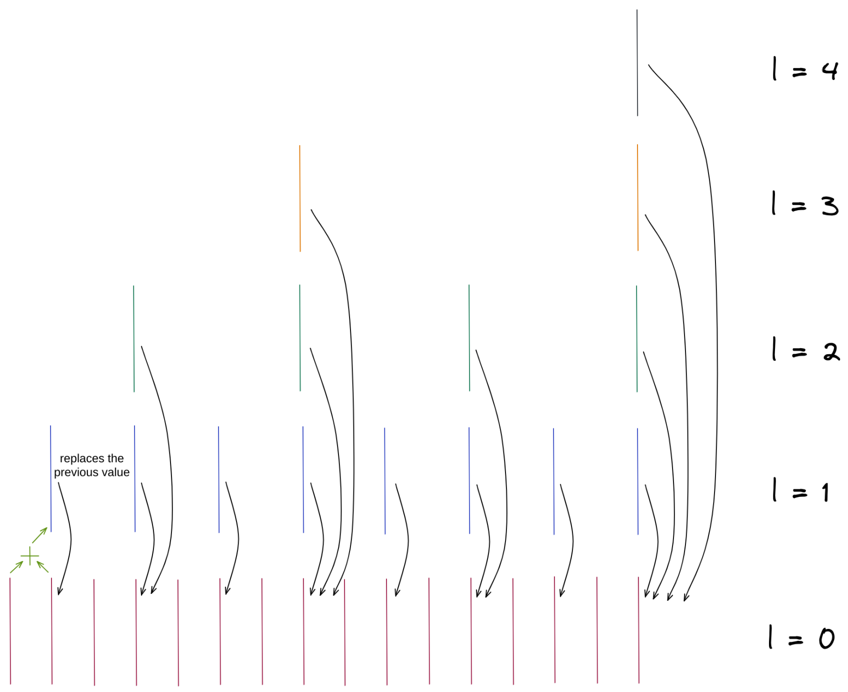

Figure 1.2: *First stage of Parallel Inclusive Scan*

The complexity is computed to be O(logn).

At the end of this stage, the items located in $p = 2^i - 1$ positions contain their final value, or, in other words, have been scanned successfully. Moreover, the values of the odd items have not been altered. The rest of the even items can be written as $2^i \cdot (2m+1), m \in \mathbb{Z}$. The following is noteworthy: we observe that at each level l of the binary tree, each child with position $p = 2^i \cdot (2m+1) - 1$, where $i \geq l$, is the sum of the items $[p - 2^l + 1, p]$. When the condition $i \geq l$ is no longer met as l increases, the position contains its final value which is equal to the sum of the items in the range $[p - 2^i + 1, p]$. For example, the element in position 11 (item 12) is the sum of the elements 9-12 by the end of the reduce stage. Additionally, we know that the items in positions $(2^i - 1)$ contain the sum of all the elements before them and themselves, so these should remain constant and can be used at the computation of the rest of the items. Thus, we now form independent binary trees that concern the elements included in the ranges $[2^i - 1, 2^{i+1} - 1]$. The goal at each level is to find the items that need only one addition to be successfully scanned. At first, these elements consist only of the item in the position $2^i - 1 + 2^{i-1}$. By adding the left limit of the range to the value of the pivot position, the latter includes the sum of all the previous items. At the next level, we observe that we can split the range in two and perform the previous operations again. We continue this process till all positions within the given range have been filled. If we visualize this, it's clear that the structure resembles the inverse binary tree of the first stage.

Hence, at each level l we divide each aforementioned range by 2^l to calculate the values of the items:

$$2^i + k \cdot \frac{(2^{i+1} - 2^i)}{2^l} = 2^i + k \cdot 2^{i-l}, \quad k = 1, ..., (2^l - 1) \in \{n \in \mathbb{Z} : n = 2m+1, m \in \mathbb{Z}\}$$

The restriction of k being odd stems from the analysis of position p, as described previously, and ensures that elements that are calculated at one level will not be re-computed at the next levels, avoiding the addition of a false offset to them.

The left limit of the range is the nearest completed item from an index. Since we split the range in two at each level, an index is at the center of its range, which can now be deducted to $[2^i - 1 + (k-1) \cdot 2^{i-l}, 2^i - 1 + (k+1) \cdot 2^{i-l}]$.

The algorithm of the second stage for a single range and level l has the form below:

```
for(k = 1; k < 2^l; k = k + 2){
    current_pos = 2^i - 1 + k * 2^(i-l);
    previous_pos = 2^i - 1 + (k-1) * 2^(i-l); // the nearest item that has been
                                              //  scanned successfully
    array[current_pos] += array[previous_pos];
}
```

A visual representation of the first two levels of this stage performed on the array used as an example in the first stage (Figure 1.2), is shown in Figure 1.3.

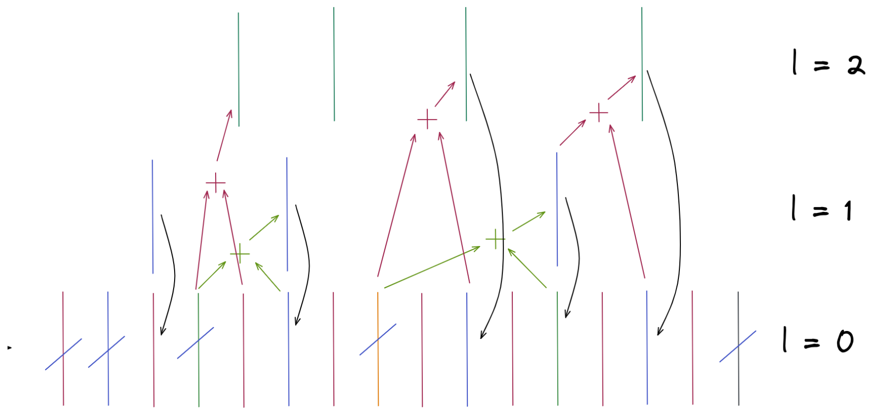

Figure 1.3: *Second stage of Parallel Inclusive Scan*

Regarding the case where the input array contains an even number of elements but doesn't satisfy the condition $n = 2^i$, a slight modification should take place. To clarify, the last range of the second stage will now be $[2^{\lfloor logn \rfloor} - 1, n-1]$. Also, in case the first division of this range results in the cursor pointing to an odd element (meaning $n = 2 \cdot (2m+1)$), we move the index to the nearest even item that fulfills the requirement of needing only one addition to be scanned successfully, which is the next to the right. This is necessary due to the odd elements not being altered during the first stage, so the method explained before for the $n = 2^i$ case would lead to false results. The rest of the operations stay as is. This alteration is shown in Figure 1.5.

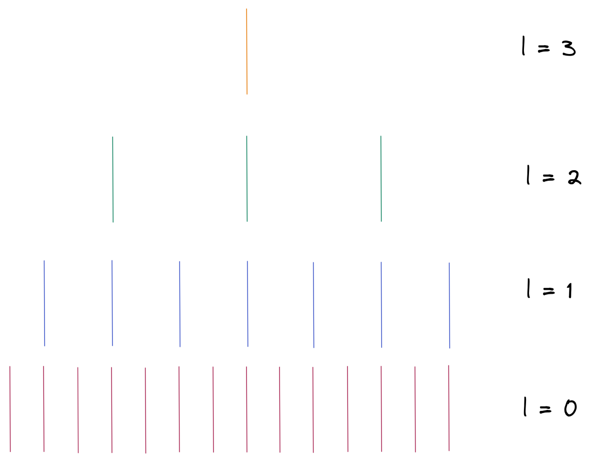

Figure 1.4: *First stage of Parallel Inclusive Scan with even, but not 2^i, items*

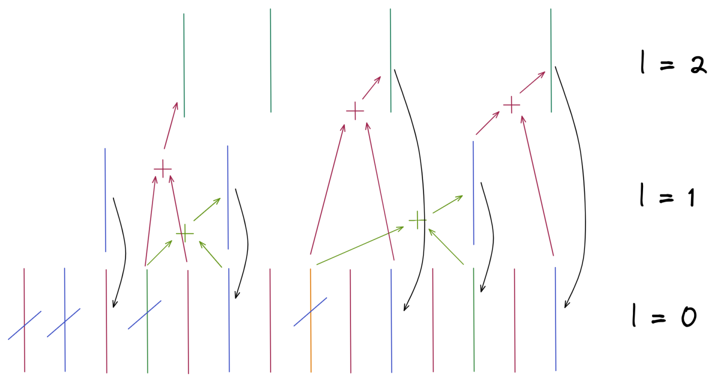

Figure 1.5: *First two levels of the second stage of Parallel Inclusive Scan with even, but not 2^i, items*

Finally, in case of an odd number of items in the input array, the algorithm applies to the range [0, n-2], and its implementation is determined by whether $log(n-1) \in \mathbb{Z}$, as described above. This exclusion of the computation of the last item occurs naturally at the first stage due to the size constraint taken into consideration when adding a 2^i offset to an even position of the array. Otherwise, we would access a memory out of bounds. On the other hand, at the second stage of the algorithm, it should be checked and excluded on purpose. After the completion of the two stages, a simple addition of the elements n-1 and n (positions n-2 and n-1 respectively) should take place. Note that this applies only to this specific case.

The complexity of the second stage is computed to be O($log(2^{i_{max}+1} - 2^{i_{max}})$) = O ($i_{max}$), since each range is processed simultaneously. Since $i+1 \leq logn$, we can instead declare the time complexity to be of O(logn).

After adding the complexities of the two stages, we end up with an overall complexity of O(logn), which is far better than the linear O(n) complexity that was observed at the sequential implementation of the `inclusive_scan`.