Single-pass Parallel Prefix Scan with Kogge-Stone and Decoupled Look-back Algorithms

Xiangyu Liu
University of Toronto

November 25, 2024

1 Introduction

This report details the implementation of a high-performance parallel prefix scan algorithm that combines the Kogge-Stone with the decoupled look-back methods. The rationale behind choosing these algorithms is discussed, along with an explanation of how they work together to optimize performance. Additionally, it provides an overview of other approaches that were attempted during development, evaluating their effectiveness and limitations. Successful techniques were retained and refined, while unsuccessful ones are analyzed to understand their shortcomings. Finally, key bottlenecks in the current implementation are identified, and potential optimizations are proposed to further enhance efficiency.

2 Algorithm Choice and Rationale

This section discusses the algorithms chosen for performing prefix scan within a block and consolidating results between blocks, along with the rationale behind these choices.

2.1 Comparison of Kogge-Stone and Brent-Kung Algorithms

In the implementation, within each block, the prefix scan is calculated using the Kogge-Stone algorithm. This algorithm is based on the scan algorithm presented by Hillis and Steele in 1986 and was later demonstrated for GPUs by Horn in 2005 [1]. The Kogge-Stone algorithm is a well-known, minimum-depth network $O(\log n)$ that uses a recursive-doubling approach to aggregate partial reductions. While its work complexity is $O(n \log n)$, the shallow depth and simple shared memory address calculations make it particularly favorable for SIMD (or SIMT) setups, such as GPU warps [2].

An alternative algorithm considered was the Brent-Kung algorithm, which is a work-efficient strategy with $2 \log n$ depth and $O(n)$ complexity. This algorithm, as presented by Blelloch in 1990, visually resembles an "hourglass" data flow [2]. The Brent-Kung approach comprises an upsweep accumulation tree, which progressively reduces parallelism, and a downsweep propagation tree, which progressively increases parallelism.

Ultimately, the Kogge-Stone algorithm was chosen over the Brent-Kung approach for several reasons:

- **Ease of Implementation:** The Kogge-Stone algorithm is relatively straightforward to implement, whereas the Brent-Kung algorithm involves more intricate index manipulations, making its implementation and debugging more complex.
- **Padding Overhead in Brent-Kung:** The Brent-Kung algorithm requires workloads to be a power of 2, necessitating padding with zeros for non-power-of-2 inputs. For instance, an input of $[1, 1, 1, 1, 1]$ needs to be padded to $[1, 1, 1, 1, 1, 0, 0, 0]$ to match the nearest power of 2 ($2^3 = 8$). After the scan, the output becomes $[1, 2, 3, 4, 5, 5, 5, 5]$, where only the first five elements $[1, 2, 3, 4, 5]$ are valid, and the remaining $[5, 5, 5]$ result from unnecessary computations. As input sizes increase, the gap to the next power of 2 widens, significantly amplifying memory usage and computational inefficiency.
- **Bank Conflicts in Brent-Kung:** The Brent-Kung algorithm suffers from shared memory bank conflicts on NVIDIA GPUs due to its memory access patterns. As multiple threads in a warp access the same memory bank, conflicts arise, causing serialized access and increasing latency. For deep binary trees, the stride between memory accesses doubles at each level, leading to increased conflicts in the middle levels of the tree, where the degree of conflicts can be as high as the warp size. These conflicts highly impact performance, particularly for larger arrays or deeper tree structures [1].

2.2 Inter-Block Aggregate Consolidation: Decoupled Look-Back

The approach used to consolidate the aggregate between blocks is the decoupled look-back method. This method, introduced by Duane Merrill and Michael Garland in 2016, is a generalization and improvement over the traditional chained-scan approach, addressing its key limitations.

The chained-scan is a single-pass parallelization method where each thread block is assigned a tile of the input. A serial dependency chain exists between thread blocks, as each block must wait for the inclusive prefix of its predecessor to be available before proceeding. This introduces latency due to inter-block communication, which limits performance. Although increasing the partition size (i.e., block workload) could reduce the number of blocks and thus mitigate some of this latency, this approach is often impractical due to shared memory limitations per thread block, since larger workloads may exceed the available on-chip resources.

The decoupled look-back method overcomes the limitations of the chained-scan approach by introducing a more flexible and parallel-friendly strategy for aggregate propagation. Each block begins by calculating its local aggregate and marking its flag as `X` to indicate that the aggregate is not yet available. Once the calculation is complete, the block updates its aggregate value and sets the flag to `A`, signaling availability.

Subsequently, the block looks back to preceding blocks to accumulate the required pre-aggregate value. During this process, if a preceding block's flag is `X`, the block continues polling until the value becomes available. If the flag is `A`, the block adds the available aggregate to its accumulated pre-aggregate and continues looking back further. If the flag is `P` (indicating a pre-aggregate is already available), the block adds the pre-aggregate value to its accumulated pre-aggregate and terminates the look-back process.

Finally, the block updates its aggregate to include the accumulated pre-aggregate and sets its flag to `P`. The accumulated pre-aggregate is then shared with all threads within the block to enable efficient propagation of the results.

This decoupled approach alleviates the strict sequential dependency between blocks, enabling more flexible parallelism and reducing latency by allowing blocks to progress dynamically as soon as the necessary data is available. It is beneficial particularly for large input sizes.

3 Design Decisions and Their Impact on Performance

This section will discuss the key design decisions made during implementation and how they improve performance.

3.1 Reverse Accumulation in Kogge-Stone

To reduce inter-block communication, the block workload was set much larger than the number of threads per block, limited by the shared memory size on NVIDIA GPUs (for example, `48 KB` or `12,288` 32-bit integers per block with a maximum of `1024` threads per block). This approach minimizes blocks but requires each thread to process multiple inputs.

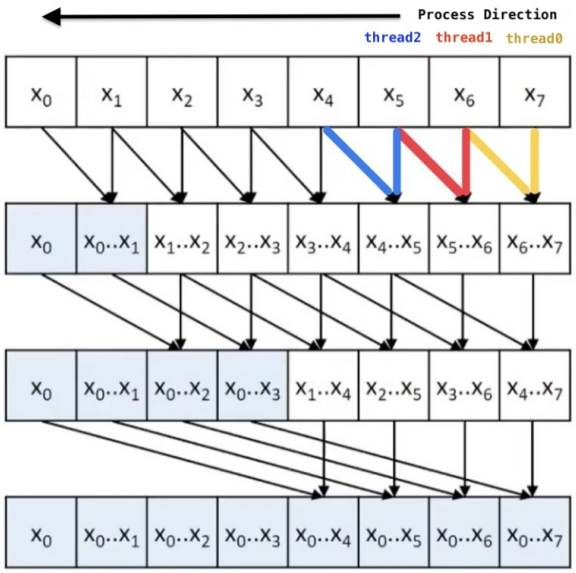

Figure 1: Reverse Accumulation method in Kogge-Stone Algorithm

A left-to-right scan risks stale reads as updated values may be read within the same round. While double buffering can address this (one for storing the original values being read and another for writing updated values), it doubles memory usage, reducing workload per block and increasing the number of blocks. I solved this issue by adopting the reverse accumulation which processes data from right to left. This ensures that updates in the current round do not interfere with reads, eliminating the need for a double buffer. It maximizes shared memory utilization, minimizes the number of blocks. For example, in Figure 1, thread 0 reads x_7 and x_6 and writes the result $x_6 + x_7$ to index 7. This update does not interfere with thread 1, which reads x_5 and x_6 and writes $x_5 + x_6$ to index 6. Similarly, thread 2 reads x_4

and x_5 and writes $x_4 + x_5$ to index 5 without being affected by the update from thread 1.

3.2 Avoiding Race Conditions with 64-bit Packing

During the decoupled look-back process, the (aggregate, flag) pair was initially stored as separate values. Without atomic writes, this caused incorrect outputs due to a time window where only one value was updated. For example, transitioning from $(2, \texttt{A})$ to $(4, \texttt{P})$ could momentarily result in an intermediate state like $(2, \texttt{P})$. If another thread reads unfortunately this inconsistent state, it would lead to calculation errors.

To resolve this, I packed the aggregate and flag into a single 64-bit integer to enable atomic updates. The first two bits represent the flag (`X-00`, `A-01`, `P-11`), while the remaining 62 bits store the aggregate. Shifting the value right by 2 extracts the aggregate, and a bitwise AND with `0b11` retrieves the flag. This encoding ensures consistency and eliminates race conditions since one update changes both the aggregate and the flag simultaneously. Here, a 64-bit integer is used instead of a 32-bit integer to prevent overflow with large input values. If the aggregate is large and occupies the 31st or 32nd bit position, it could lose meaningful bits during shifts in a 32-bit system. The 64-bit format provides sufficient space, and ensures correctness for large input values.

4 Exploration of Alternative Approaches

This section discusses alternative approaches that were explored during the development process, analyzing the reasons for rejection.

4.1 Brent-Kung Algorithm

One of the initial approaches implemented was the Brent-Kung algorithm. While efficient, it requires input sizes to be powers of 2, necessitating zero-padding for other sizes. This padding introduces wasteful computations, increases shared memory usage, and causes much more blocks. The resulting higher block count amplifies inter-block communication latency, and makes it less suitable for single-pass methods like chained-scan or decoupled lookback.

4.2 Chained-Scan Implementation

Before adopting the decoupled lookback method, I initially implemented a chained-scan approach. While functional, it was heavily impacted by the number of blocks required. For an input size of `100,000,007u` and a maximum workload of `12,288` elements per block, approximately `8,139` blocks were needed. This resulted in `8,138` aggregate handoffs, significantly affecting performance.

Testing various configurations of block workload and threads per block, the combination of Kogge-Stone within blocks and chained-scan between blocks achieved a speedup of around 30×. To address the inefficiency in aggregate propagation, I adopted the decoupled look-back method, which mitigated these bottlenecks and achieved an improved speedup of 42×.

5 Performance Bottlenecks and Potential Optimization

The current implementation achieves great speedup but still faces bottlenecks. A key issue is the global memory polling during the decoupled look-back phase, where a designated thread in each block polls a global memory array for aggregate values. This reliance on slower global memory becomes a limitation as the number of blocks increases.

One optimization could involve addressing the 2^n-aligned input size limitation of the Brent-Kung algorithm, enabling its integration with the decoupled lookback method. Brent-Kung computes the aggregate at the end of its upsweep phase, allowing aggregates to be passed to subsequent blocks before the downsweep phase begins. This early availability reduces aggregate read latency between blocks. Furthermore, the accumulated aggregate could be merged at the start of the downsweep phase, which eliminates the need for a final propagation phase where each thread updates its value with the accumulated aggregate. This optimization would decrease the number of writes by n, further enhancing efficiency.

References

[1] Mark Harris, Shubhabrata Sengupta, and John D. Owens, *GPU Gems 3*. Chapter 39, pages 851–876. Addison-Wesley, 2007. Available at:`https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda`.

[2] Duane Merrill and Michael Garland, "Single-pass Parallel Prefix Scan with Decoupled Look-back," *NVIDIA Technical Report NVR-2016-002*, March 2016.