Parallel Approach to Sliding Window Sums

Roman Snytsar\(^1\) and Yatish Turakhia\(^2\)

\(^1\)Microsoft Corp., One Microsoft Way, Redmond WA 98052, USA.
\(^2\)Stanford University, Stanford CA 94305, USA.

Abstract

Sliding window sums are widely used in bioinformatics applications, including sequence assembly, k-mer generation, hashing and compression. New vector algorithms which utilize the advanced vector extension (AVX) instructions available on modern processors, or the parallel compute units on GPUs and FPGAs, would provide a significant performance boost for the bioinformatics applications.

We develop a generic vectorized sliding sum algorithm with speedup for window size \(w\) and number of processors \(P\) is \(O(P/w)\) for a generic sliding sum. For a sum with commutative operator the speedup is improved to \(O(P/\log(w))\). When applied to the genomic application of minimizer based k-mer table generation using AVX instructions, we obtain a speedup of over 5x.

For a family of sliding sums that allow recurrent interpretation we achieve speedup that is independent of \(w\): \(O(P/\log(P))\). With ntHash algorithm rewritten using AVX instructions we have observed a speedup close to 3x.

1 Introduction

Bioinformatics algorithms for sequence assembly, indexing, search, and compression evolve at a breakneck rate. Still, many foundational ideas, like using short substrings known as k-mers, hashing schemas, and bespoke indexing mechanisms, stay relevant. A relatively new idea quickly gaining popularity is the use of minimizers [Roberts et al., 2004]. Minimizers exploit the sequence contiguity allowing to represent the sequence with a smaller number of k-mers, thus producing more compact indices. Minimizers have been successfully used for k-mer counting [Deorowicz et al., 2015], sequence alignment [Li, 2018], and indexing [Wood and Salzberg, 2014]. While some research has been directed towards improving the performance of minimizers [Marçais et al., 2017], little attention has been paid to the properties of the underlying algorithm of sliding window minimum.

In this paper we explore the properties of generic sliding window sums and uncover the potential for parallel speedup. We show that the sliding sum approach could be extended beyond minimizers to k-mer generation and hashing, and develop fast vector implementations not only for minimizer generation but also for k-mer hashing.

2 Methods

2.1 Prefix Sum

Parallel algorithms are often constructed from a set of universal building blocks. One of the hardest to identify, but extremely useful is the concept of a prefix sum, and the accompanying scan algorithm. A prefix sum is a transformation that takes an operator \(\oplus\), and a sequence of elements

\[x_0, x_1, \ldots, x_i, \ldots\]

and returns the sequence

\[y_i = \sum_{j=0}^{i} x_j = x_0 \oplus x_1 \oplus \ldots \oplus x_i\]  

(1)

or in recurrent form

\[y_i+1 = y_i \oplus x_{i+1}\]  

(2)

Despite the data carry dependency, the first \(N\) elements of the prefix sum with an associative operator could be computed in \(O(\log(N))\) parallel steps using scan algorithm, as shown by [Blelloch, 1993].

2.2 Sliding Window Sum

Sliding window sum (sliding sum) takes a window size \(w\) and an operator \(\oplus\), and a sequence of elements, and returns the sequence

\[y_i = \sum_{j=i}^{i+w-1} x_j = x_i \oplus x_{i+1} \oplus \ldots \oplus x_{i+w-1}\]  

(3)

where each sum is defined in terms of the operator \(\oplus\) and contains exactly \(w\) addsends. The asymptotic complexity of a naive sliding sum algorithm is \(O(wN)\) where \(N\) is the length of the source sequence.

It is worth mentioning that every sum defined by Equation (3) is a prefix sum with operator \(\oplus\) and input sequence \(x_i \ldots x_{i+w-1}\). Many useful operators are associative, so the prefix scan algorithm is applicable here, reducing complexity in this case to \(O(N\log(w))\) parallel steps. We have observed this optimization being implemented by the vectorizing compilers. It is, however, possible to further improve performance by exploiting more subtle operator properties.

Starting with a string \(R\) over an alphabet \(\Sigma R = r_0, r_1, \ldots, r_n, r_f \in \Sigma\), we can define k-mers of \(R\) as a sliding sum over window size \(k\), string concatenation operator, and \(R\). If we introduce operation \(\text{substr}(K, i)\), a substring of string \(K\) starting from position
2.3 Vector Algorithms

K-mers are widely used for string searches, in particular in the gene mapping tools like BLAST [Altschul et al., 1990], Snap [Zaharia et al., 2011], FreeBayes [Garrison and Marth, 2012]. Often the algorithms work not with the k-mers directly but with the hash values of k-mers. [Mohamadi et al., 2016] proposed nHash, a variation of the fingerprint schema by [Karp and Rabin, 1987]. String characters are replaced by the elements of the array of random numbers \( h_i = h[r_i] \), and the hash formula is expressed in terms of \( h_i \).

\[
H_i = (h_i \gg k - 1) \land (h_{i+1} \gg k - 2) \land \cdots \land h_{i+k-1}
\]

where \( \gg \) is a k-bit barrel right-shift operator, and \( \land \) is an exclusive OR. Both operators are the reverse of themselves \((a \land b) \land c = a \land (b \land c)\). and nHash could be expressed recursively using only \( \gg \) and \( \land \):

\[
H_{i+1} = (H_i \gg 1) \land h_i \land h_{i+k}.
\]

Another advantage of the nHash is that it works with the source sequence bypassing the k-mer construction.

Further reduction of the memory requirements for genome indexing, search and assembly is linked to the notion of the minimizers, the sliding sum over window size \( w \), operator \( \min \), and the sequence of hashed k-mers. The reverse operator trick does not work with \( \min \), however, it is possible to achieve similar \( O(N) \) complexity at the cost of using elaborate queue-based data structures [Tangwongsan et al., 2015].

We will now take a closer look at the connection between sliding sums and prefix sums, and attempt to supersede the linear complexity achieved by the aforementioned approaches.

2.3 Vector Algorithms

Our first algorithm is a vector-friendly way of calculating sliding sum assuming the input sequence elements become available one by one:

\[
Y \leftarrow (\sum_{j=0}^{w-2} x_j, \sum_{j=1}^{w-2} x_j, \ldots, x_{w-3} \oplus x_{w-2}, x_{w-2}, 0, \ldots, 0)\]

for \( i = w - 1 \) to \( N \)

\[
X \leftarrow (x_1, x_2, \ldots, x_i, 0, \ldots, 0)\]

\[
Y \leftarrow Y + X
\]

\[
y_i \leftarrow Y[i\mod w]
\]

\[
Y \leftarrow Y \ll (p - w)
\]

end for

Listing 1: Scalar Input.

Vector Y is initialized to the suffix sums with the number of elements decreasing from \( w - 1 \) to 0. Then in a loop every incoming element \( x_k \) is broadcast to the first \( w \) elements of vector X. After vector addition the zeroth element of Y contains the next sliding sum. Next, the vector Y is shifted left by one element, as denoted by operator \( \ll \), and the state is ready for the next iteration. Asymptotic complexity of the scalar input algorithm is \( O(N) \) with no additional requirements on the operator \( \oplus \).

This result could be improved if we assume that the input sequence arrives packed in vectors of width \( P > w \).

\[
Y \leftarrow \left( \sum_{j=0}^{w-2} x_j, \sum_{j=0}^{w-3} x_j, x_0 \oplus x_1, x_0, 0, \ldots, 0 \right)
\]

for \( k = w - 1 \) to \( N \) step \( P \)

\[
X \leftarrow (x_k, x_{k+1}, \ldots, x_{k+P-1})
\]

\[
X1 = (x_0, x_0 \oplus x_1, \ldots, \sum_{j=0}^{w-2} x_j, \sum_{j=0}^{w-1} x_j, \ldots, \sum_{j=0}^{P-1} x_j)
\]

\[
Y1 = \left( 0, \ldots, 0, \sum_{j=p\mod w}^{w-2} x_j, \sum_{j=p\mod w}^{P-1} x_j, \ldots, \sum_{j=p\mod w}^{w-1} x_j \right)
\]

\[
Y \leftarrow Y \oplus X1
\]

\[
y_k \ldots y_{k+P-1} \leftarrow Y[0] \ldots Y[p - 1]
\]

\[
Y \leftarrow Y1 \ll (P - w)
\]

end for

Listing 2: Vector Input.

At every iteration \( P \) input elements are placed into vector \( X \). \( X1 \) is filled with the prefix sums of up to \( w \) addends, and \( Y1 \) is filled with the suffix sums constructed from the elements of \( X \). Then the vector sum of \( Y \) and \( X1 \) yields the next \( P \) output elements. Finally, the suffix sums from \( Y1 \) are shifted into proper positions in vector \( Y \), and it is ready for the next iteration. The asymptotic speedup thus is \( O(P/w) \) for any operator \( \oplus \). If \( \oplus \) is associative, the prefix/suffix sums could be computed in parallel using the algorithm in [Billeloch, 1993], and the complexity reduces to \( O(P/(wlog(w))) \).

For example, since \( \min \) is an associative operator, the sliding window minimum can be computed using the faster version of the vector input algorithm.

2.4 nHash

Before we could apply the same algorithm to nHash, we have to show that Equation [5] defines a sliding window sum, and prove the operator associativity.

Consider the sequence

\[
y_i = \begin{cases} h_0, & i = 0 \\ (y_{i-1} \gg a_i) \land h_i, & i > 0 \end{cases}
\]

Operators \( \gg \) and \( \land \) have the following properties:

1. XOR is associative:

\[
(a \land b) \land c = a \land (b \land c)\]
We can define a new operator $\gg$.

$$ (a \gg b) \gg c = (a \gg c) \gg (b \gg c). \quad (11) $$

We can now define a sliding window sum with a window size $w$, associative operator $\bullet$, and input sequence $z_i = (a_i)_{h_i}$.

$$ z_0 = \left( \begin{array}{c} 0 \\ h_0 \end{array} \right); z_i = \left( \begin{array}{c} a_i \\ h_i \end{array} \right) \quad (15) $$

The second component of the pair $z_i$ generates ntHash by definition of the operator $\bullet$. If $a_i = 1$, then the first component of the pair is a sequence index and is stored implicitly as an array position. Thus the fast vector input algorithm could be used for ntHash.

Better yet, the recurrent form of ntHash as described by Equation 8 is a prefix sum with an associative operator $\bullet$ over the sequence $h'_i = h_{i+k} \land h_i$, and could be computed in $O(\log(N))$ time.

This observation could be extended to the bigger family recursive sliding sums. If operator $\oplus$ is associative and has the reverse operator $\ominus$ so that the sliding sum could be rewritten in the recursive form

$$ y_{i+1} = y_i \oplus (x_{i+w} \ominus x_i), \quad (16) $$

then the sliding sum is a prefix sum with operator $\oplus$ and input sequence $z_i = x_i \ominus x_{i-w}$, and could be accelerated by $O(P/\log(P))$ regardless of $w$.

This reduction to prefix sum is useful for small $P$, for example, when using the CPU-based vector instructions. Or, conversely, when $w$ is prohibitively large, like in the rolling sum with $w = 8192$ as implemented in the Lempel-Ziv algorithm [Ziv and Lempel, 1977].

2. Shift is distributive across XOR:

$$ (a \land b) \gg c = (a \gg c) \land (b \gg c). \quad (11) $$

3. Shift is semi-associative with a companion operator $\oplus$, i.e.

$$ (a \gg b) \gg c = a \gg (b + c). \quad (12) $$

We can now define a new operator $\bullet$ acting on pairs $z_i = (a_i)_{h_i}$

$$ \left( \frac{a_i}{h_i}, \frac{a_j}{h_j} \right) \bullet \left( \frac{a_i + a_j}{h_i \gg a_j \land h_j} \right) = \left( \frac{a_k}{h_k} \right) \cdot \left( \frac{a_i}{h_i} \right) \cdot \left( \frac{a_j}{h_j} \right), \quad (13) $$

Given the properties of operators $\gg$ and $\land$ we can prove that $\bullet$ is associative:

$$ \left( \left( \left( \frac{a_i}{h_i} \bullet \frac{a_j}{h_j} \right) \bullet \frac{a_k}{h_k} \right) \bullet \frac{a_l}{h_l} \right) = \left( \frac{a_i + a_j}{h_i \gg a_j \land h_j} \right) \bullet \frac{a_l}{h_l} = \left( \frac{a_i + (a_j + a_k)}{(h_i \gg (a_j \land h_k)} \right) \land (h_j \gg a_k) \land h_k) \right) \bullet \frac{a_l}{h_l} = \left( \frac{a_i + a_j}{h_i \gg a_k} \right) \land (h_j \gg a_k) \land h_k) \right) \bullet \frac{a_l}{h_l} = \left( \frac{a_i}{h_i} \right) \bullet \left( \frac{a_j}{h_j} \right) \bullet \left( \frac{a_k}{h_k} \right), \quad (14) $$

We can now define a sliding window sum with a window size $w$, associative operator $\bullet$, and input sequence $z_0 = \left( \begin{array}{c} 0 \\ h_0 \end{array} \right); z_i = \left( \begin{array}{c} a_i \\ h_i \end{array} \right)$.

The second component of the pair $z_i$ generates ntHash by definition of the operator $\bullet$. If $a_i = 1$, then the first component of the pair is a sequence index and is stored implicitly as an array position. Thus the fast vector input algorithm could be used for ntHash.

Better yet, the recurrent form of ntHash as described by Equation 8 is a prefix sum with an associative operator $\bullet$ over the sequence $h'_i = h_{i+k} \land h_i$, and could be computed in $O(\log(N))$ time.

This observation could be extended to the bigger family recursive sliding sums. If operator $\oplus$ is associative and has the reverse operator $\ominus$ so that the sliding sum could be rewritten in the recursive form

$$ y_{i+1} = y_i \oplus (x_{i+w} \ominus x_i), \quad (16) $$

then the sliding sum is a prefix sum with operator $\oplus$ and input sequence $z_i = x_i \ominus x_{i-w}$, and could be accelerated by $O(P/\log(P))$ regardless of $w$.

We tested the performance of various sliding minimum algorithms using the hashed 15-mers of the reference human genome assembly (GRCh38) from the Genome Reference Consortium. The test imitates a minimizer based seed table construction by a long-read aligner, such as Minimap2 [Li, 2018], GraphMap [Sović et al., 2016] or Darwin [Turakhia et al., 2018]. Figure 1 compares the performance of the naive array-based algorithm, linear dequeue-based algorithm, and our proposed vector algorithm.

Deque-based algorithm performance is indeed independent of the window size. It comes, however, at the cost of a significant overhead of managing the deque data structure and unpredictable branching.

Array-based algorithm, despite the worst asymptotic complexity, is simple to implement, and benefits from the automatic compiler vectorization. It is clear how the times drop when the window size is aligned with the SIMD vector width (8 and 16). For small window sizes the array algorithm is competitive with the deque approach.

Our vector sliding sum algorithm beats both previous implementations by a factor of 5. It demonstrates an even better performance as the vector width increases, matching the theoretical performance estimate.

Additionally, we have implemented vector ntHash algorithm that computes 4 values (AVX) or 8 values (AVX512) in a single pass. Vector implementations have been tested on the complete GRCh38 genome as well and the results are shown in Figure 2. As expected, the 4-vector version is 1.7 times faster than the baseline scalar implementation by [Mohamadi et al., 2016], close to the theoretical speedup of 2. The 8-vector version outperforms scalar by the factor of 2.5 with the theoretical speedup of 3.
4 Conclusion

We introduced a family of algorithms for parallel evaluation of sliding window sums. The parallel speedup for window size \( w \) and number of processors \( P \) is \( O(P/w) \) for a generic sliding sum. For a sum with a commutative operator the speedup is improved to \( O(P/\log(w)) \). For a family of sliding sums that allow recurrent interpretations, the speedup is independent of \( w \): \( O(P/\log(P)) \). This gives the developer a choice of fast branch-less algorithms suitable for implementation on any modern parallel architecture including modern CPUs with instruction-level parallelism, pipelined GPUs, or FPGA reconfigurable hardware.

While we concentrate on the faster sliding window sum algorithms for bioinformatics, our findings are relevant for accelerating all the numerous sliding window applications from the Lempel-Ziv compression [Ziv and Lempel, 1977] to high frequency data mining [Ikonomovska et al., 2007]

**References**

[Altschul et al., 1990] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410.

[Blelloch, 1993] Blelloch, G. E. (1993). Prefix sums and their applications. In *Synthesis of Parallel Algorithms*. Morgan Kaufmann.

[Deorowicz et al., 2015] Deorowicz, S., Kokot, M., Grabowski, S., and Debudaj-Grabysz, A. (2015). Knm2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576.

[Garrison and Marth, 2012] Garrison, E. and Marth, G. (2012). Haplotype-based variant detection from short-read sequencing. *arXiv preprint arXiv:1207.3907*.

[Ikonomovska et al., 2007] Ikonomovska, E., Loskovska, S., and Gjorgievik, D. (2007). A survey of stream data min-

![Figure 2: Performance of the parallel ntHash.](image-url)