Vectorized Character Counting for Faster Pattern Matching

Roman Snytsar
Microsoft Corp., One Microsoft Way, Redmond WA 98052, USA
Roman.Snytsar@microsoft.com

Keywords: Parallel Processing, Vectorization, Bioinformatics, FM-Index.

Abstract: Many modern sequence alignment tools implement fast string matching using the space efficient data structure called FM-index. The succinct nature of this data structure presents unique challenges for the algorithm designers. In this paper, we explore the opportunities for parallelization of the exact and inexact matches and present an efficient SIMD solution for the Occ portion of the algorithm. Our implementation computes all eight Occ values required for the inexact match algorithm step in a single pass. We showcase the algorithm performance in a multi-core genome aligner and discuss effects of the memory prefetch.

1 INTRODUCTION

FM-index has been developed as a space efficient index for string matching. The backward search over the index finds exact matches of a pattern in time that is linear relative to the length of the pattern regardless the size of the reference. Even though the applications are numerous, for instance text compression (Navarro and Mäkinen, 2007), and indexing (Zhang et al., 2013a), FM-index has become especially popular with the developers of DNA sequence aligners like Bowtie (Langmead et al., 2009), SOAPv2 (Li et al., 2009), and BWA (Li and Durbin, 2009). Next, we introduce the fundamentals of FM-index construction and operation.

2 BACKGROUND

Let \( R \) be a string of length \( n \) over some alphabet \( \Sigma \). A special character $ that is not part of the alphabet and is lexicographically smaller than any character in \( \Sigma \) is appended to the end of the string. \( R[i] \) denotes a character in \( R \) at position \( i \), and \( R[i, j] \) is a substring of \( R \) ranging from \( i \) to \( j \). Suffix array \( SA(R) \) is then defined as an integer array containing starting positions of all suffixes of \( R \) in a sorted order so that

\[
R[SA[i] - 1, n] < R[SA[i], n], \quad 1 < i < n \quad (1)
\]

Suffix array could be constructed by simply sorting all suffixes of a string. More sophisticated algorithms take into account the fact that all strings are related to each other and achieve much better asymptotic complexity and practical performance (Puglisi et al., 2007).

\[
\begin{array}{cccccc}
\text{SA} & \text{BWT} & \text{Occ} \\
4 & $ & $ACA$ & $G & 0 & 0 & 1 & 0 \\
0 & ACAG$ & ACAG & $ & 0 & 0 & 1 & 0 \\
2 & AG$ & AG$A & C & 0 & 1 & 1 & 0 \\
1 & CAG$ & CAG$ & A & 1 & 1 & 1 & 0 \\
3 & G$ & G$AC & A & 2 & 1 & 1 & 0 \\
\end{array}
\]

Next, for each character \( b \) in \( \Sigma \) and for every \( 0 \leq k < n \) we record the number of occurrences of \( b \) in the BWT substring \( BWT[0, k] \), and store it in the table \( \text{Occ} \). Additionally, we store the total of occurrences of all characters lexicographically preceding \( b \) in \( BWT \) into the table \( C \). It is easy to compute \( C \) as an exclusive prefix sum of the last row of \( \text{Occ} \). For any
single character at position $i$ in a pattern $W$, the interval of rows in the BWM starting with this character is
easily computed from $C^∗$:

$$\{k,l\} = \{C[W[i]], C[W[i] + 1]\}$$  \hspace{1cm} (3)

From this initial interval, it is possible to extend the search backward from the starting position using the following recursive procedure:

```
ExactRec(W, i, k, l)
if $i < 0$ then
    return $\emptyset$
if $i < 0$ then
    return $\{k,l\}$
l ← $l \cup$ ExactRec(W, i − 1, $\Psi^{-1}(W[i])$
for each $b \in \Sigma$ do
    $k ← C(b) + Occ(b, k-1) + 1$
l ← $C(b) + Occ(b, l)$
if $k ≤ l$ then
    $l ← l \cup$ ExactRec(W, i − 1, $\Psi^{-1}(W[i])$
if $b = W[i]$ then
    $l ← l \cup$ ExactRec(W, i − 1, $\Psi^{-1}(W[i])$
else
    $l ← l \cup$ ExactRec(W, i − 1, $\Psi^{-1}(W[i])$
return $l$
```

Listing 1: Backward Exact Match.

After the search is complete, the final BWT interval is mapped back to the locations in reference using the suffix array.

BWA ([Li and Durbin, 2009]) extends this algorithm to allow a predetermined number of mismatches $z$:

```
InexRec(W, i, z, k, l)
if $z < 0$ then
    return $\emptyset$
if $i < 0$ then
    return $\{k,l\}$
l ← $l \cup$ InexRec(W, i − 1, $\Psi^{-1}(W[i])$
for each $b \in \Sigma$ do
    $k ← C(b) + Occ(b, k-1) + 1$
l ← $C(b) + Occ(b, l)$
if $k ≤ l$ then
    $l ← l \cup$ InexRec(W, i − 1, $\Psi^{-1}(W[i])$
if $b = W[i]$ then
    $l ← l \cup$ InexRec(W, i − 1, $\Psi^{-1}(W[i])$
else
    $l ← l \cup$ InexRec(W, i − 1, $\Psi^{-1}(W[i])$
return $l$
```

Listing 2: Inexact Match.

Note that every step of the inexact match algorithm consumes eight Occ values compared to two Occ values required by the exact match. In theory, all Occ values could be precomputed, but holding the full Occ array for the human genome reference would consume approximately 100GB of memory. To save memory space, the FM-index over the DNA alphabet is often stored using a cache-friendly approach introduced in ([Gog and Petr, 2014]), that harkens back to the bucket layout from the original FM-index paper ([Ferragina and Manzini, 2000]). Values of Occ($*,k$) for every $k$ that is a multiple of 128 are stored in memory followed by 128 characters of BWT in 2-bit encoding. Four Occ counters occupy 256 bits, as does the BWT string. Such data arrangement aligns well with AVX vector operations. For Occ counters that are not at the factor of 128 positions, the values must be calculated on the fly. Furthermore, suffix array is compressed it a similar manner. Only values of SA[$k$] where $k$ is a multiple of 32 are stored in memory, while all the values in between are recomputed using the Inverse Suffix Array relationships:

$$Ψ^{-1}(i) = C[BWT[i]] + Occ(BWT[i], i)$$  \hspace{1cm} (4)

$$SA[k] = SA[(Ψ^{-1})(k)] + j$$  \hspace{1cm} (5)

It means that the Equation 4 is applied over and over until for some $j$ the result comes out to be a multiple of 32, and the SA value could be constructed according to Equation 5.

Even though the memory saving measures do not change the asymptotic complexity of the match algorithm, in reality they add hundreds of computations of Occ to every search. Given that the search is performed multiple times for each and every read out of billions required for the alignment of a human genome, Occ function performance becomes crucial.

3 SOLUTION

Our approach to computing Occ could be traced back to the algorithm by ([Vigna, 2008]), that performs memory table lookups to count character occurrences in each byte of the BWT string. We replace the memory lookups with the half-byte register lookups, building on an idea first proposed by Mula for the bit population count ([Mula et al., 2017]). Note however that we do not attempt to reduce the character counting problem to bit counting, and apply the half-byte technique directly to the BWT string.

The input BWT string is masked with zeros beforehand for situations when the character at position $k$ is in the middle of the byte. The result of 256-bit occurrence count should be corrected for the extra $127 – k$ A characters.

3.1 Lookup

Every byte in the BWT string is split into its higher and lower half. Since each half byte value cannot be greater than 15, the lookup values now fit into a single vector register and could be retrieved via the VP-SHUF instruction. The lookup returns all four counters in a 2-bit format packed into a byte. Two bits are sufficient as a half byte contains just two characters. Additionally, the OccLo result is pre-converted into its bit complement to assist with subsequent extraction operation.
Figure 2: Example of counting letter ‘g’ in a 32-character portion of BWT string.

3.2 Extraction

After the lookup phase, all counters are tightly packed into two vector registers. Any addition operation could result in overflowing the 2-bit values. Before proceeding we have to extract counters for a given character X.

Both OccLo and OccHi are shifted to bring OccX into the two lower bits of the byte. High bits of OccLo are then filled with ones and high bits of OccHi with zeros. At this point OccHi contains unsigned byte values of OccHiX and OccLo contains values of 255 – OccLoX. The merged values are then fed into the VPSADBW instruction. It sums the absolute differences of eight consecutive bytes and stores the result as a 64-bit integer. At the end of this operation the result vector contains four partial sums in the form of 2040 – Σ(OccLoX + OccHiX). The final horizontal sum yields 8160 – OccX, and the final subtraction is combinable with the correction for the extra A count.

3.3 Aggregation

The extraction sequence runs four times to collect partial sums for characters ACGT, CAGT, TGCA, and GTAC in four vector registers. Aggregating four final sums within a single register then takes just three additions and three shuffle operations, only one of which crosses the 128-bit lane boundary. The absence of data dependencies between extraction operations facilitates efficient use of the SIMD pipeline and keeps all arithmetic ports busy. Outputs for all eight counters required for one step of the inexact search fit into a single AVX512 register and are computed in one vector pass.

4 IMPLEMENTATION

We have implemented the half-byte Occ algorithm using the AVX2 instruction set. The AVX512 version computing 8 values in parallel has also been integrated in the inexact search algorithm. The assembly code along with the Intel Architecture Code Analyzer throughput report is listed in the Appendix. The code is indeed well balanced across the ports but is expected to bottleneck on the backend meaning that the memory access pattern is crucial for the real word performance.

4.1 Experimental Setup

The computer platform is an Intel Xeon Platinum 8168 system with 16 cores running at 2.7 GHz and 32GB of RAM. To test the software performance we
have run the BWA alignment tool with 16 threads (-t 16) on a 30X Human genome sample NA12878 from the 1000 Genomes database using hg38 as a reference. We have executed the BWA version 7.15 to establish the baseline, and then replaced the Occ code with our AVX2 and AVX512 implementations. The total runtimes got collected from the BWA reports, and the percentage of time spent in the BWA and SA code measured via profiling.

4.2 Results

The vectorized SA code runs twice as fast as the scalar version, mostly due to the predictable memory prefetch pattern. The BWT code exhibits only a 30% speedup, and the switch to AVX512 does not bring any gains. At this point the BWA code is completely memory bound. To utilize the vectorization performance gains completely, we would have to explore ideas for improving the cache locality of the FM-index (Chacon et al., 2013; Zhang et al., 2013b). Despite the memory bottlenecks, the overall runtime is improved by 25%.

REFERENCES

Chacon, A., Moure, J. C., Espinosa, A., and Hernandez, P. (2013). n-step fm-index for faster pattern matching. *Procedia Computer Science*, 18:70 – 79. 2013 International Conference on Computational Science.

Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE.

Gog, S. and Petri, M. (2014). Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314.

Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. L. (2009). Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):R25.

Li, H. and Durbin, R. (2009). Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760.

Li, R., Yu, C., Li, Y., Lam, T.-W., Yiu, S.-M., Kristiansen, K., and Wang, J. (2009). Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967.

Mula, W., Kurz, N., and Lemire, D. (2017). Faster population counts using avx2 instructions. The *Computer Journal*, 61(1):111–120.

Navarro, G. and Mäkinen, V. (2007). Compressed full-text indexes. *ACM Comput. Surv.*, 39(1).

Puglisi, S. J., Smyth, W. F., and Turpin, A. H. (2007). A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2).

Vigna, S. (2008). Broadword implementation of rank/select queries. In *International Workshop on Experimental and Efficient Algorithms*, pages 154–168. Springer.

Zhang, D., Liu, Q., Wu, Y., Li, Y., and Xiao, L. (2013a). Compression and indexing based on bwt: A survey. In *2013 10th Web Information System and Application Conference*, pages 61–64.

Zhang, J., Lin, H., Balaji, P., and Feng, W.-c. (2013b). Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures. In *Cluster, Cloud and Grid Computing (CCGRID)*, 2013 13th *IEEE/ACM International Symposium on*, pages 377–384. IEEE.

APPENDIX

Intel Architecture Code Analyzer report for the AVX implementation of ψ⁻¹.
Throughput Analysis Report

Block Throughput: 20.63 Cycles
Throughput Bottleneck: Backend

Loop Count: 22

Port Binding In Cycles Per Iteration:

| Port | 0 - DV | 1 | 2 - D | 3 - D | 4 | 5 | 6 | 7 |
|------|--------|---|-------|-------|---|---|---|---|
| Cycles | 16.3 | 0.0 | 16.4 | 7.5 | 7.5 | 7.5 | 7.5 | 0.0 | 16.3 | 1.0 | 0.0 |

| Num Of | Ports pressure in cycles | |
|--------|---------------------------|---|
| Uops   | 0 | 1 | 2 - D | 3 - D | 4 | 5 | 6 | 7 |
| 2^ | | 1.0 | 0.5 | 0.5 | 0.5 | 0.5 | | |
| 1 | | 0.5 | 0.5 | 0.5 | 0.5 | | |
| 0.5 | | 0.5 | 0.5 | 0.5 | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.7 | | 0.7 | 0.7 | | | | |
| 1.0 | | 1.0 | 1.0 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
| 0.3 | | 0.3 | 0.3 | | | | |
vpsadbw ymm1, ymm3, ymm5

vpadd ymm5, ymm8, ymmword ptr [rip+0x21dffe]

vpermilpd ymm0, ymm1, 0x5

vpaddq ymm4, ymm0, ymm1

vpaddd ymm5, ymm8, ymmword ptr [rip+0x21debc]

vpermq ymm0, ymm4, 0x4e

vpaddq ymm1, ymm0, ymm4

vmovq r8, xmm0

vpsubq ymm2, ymm3, ymm1

vpcmpeqq xmm1, xmm10, xmm0

vpandn xmm0, xmm1, xmm4

vmovq r8, xmm0

Total Num Of Uops: 65

Analysis Notes:

Backend allocation was stalled due to unavailable allocation resources.