ISTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

# OPTIMIZING PACKED STRING MATCHING On AVX2 PLATFORM

M.Sc. THESIS

Mehmet Akif AYDOĞMUŞ

Department of Computational Science and Engineering

Computational Science and Engineering Master Programme

DECEMBER 2018

ISTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

# OPTIMIZING PACKED STRING MATCHING On AVX2 PLATFORM

M.Sc. THESIS

Mehmet Akif AYDOĞMUŞ
(702121013)

Department of Computational Science and Engineering

Computational Science and Engineering Master Programme

Thesis Advisor: Assoc. Prof. Dr. M. Oğuzhan KÜLEKCİ

DECEMBER 2018

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ BİLİŞİM ENSTİTÜSÜ

# AVX2 PLATFORMU ÜZERİNDE PAKETLENMİŞ DİZGİ EŞLEŞTİRME VE OPTİMİZASYONU

YÜKSEK LİSANS TEZİ

**Mehmet Akif AYDOĞMUŞ**
**(702121013)**

**Hesaplamalı Bilim ve Mühendislik Anabilim Dalı**

**Hesaplamalı Bilim ve Mühendislik Yüksek Lisans Programı**

**Tez Danışmanı: Assoc. Prof. Dr. M. Oğuzhan KÜLEKCİ**

**ARALIK 2018**

Mehmet Akif AYDOĞMUŞ, a M.Sc. student of ITU Informatics Institute Engineering and Technology 702121013 successfully defended the thesis entitled “OPTIMIZING PACKED STRING MATCHING On AVX2 PLATFORM”, which he/she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

| | | |
|---|---|---|
| **Thesis Advisor :** | **Assoc. Prof. Dr. M. Oğuzhan KÜLEKCİ**<br>Istanbul Technical University | .............................. |
| **Jury Members :** | **Prof. Dr. Mustafa Ersel KAMAŞAK**<br>Istanbul Technical University | .............................. |
| | **Assoc. Prof. Dr. M. Oğuzhan KÜLEKCİ**<br>Istanbul Technical University | .............................. |
| | **Assoc. Prof. Dr. Gökhan Bilgin**<br>Yıldız Technical University | .............................. |

**Date of Submission : 16 November 2018**
**Date of Defense : 12 December 2018**

*To my family,*

## FOREWORD

Firstly, I would like to express my most sincere gratitude to my supervisor Assoc. Prof. Dr. Muhammed Oğuzhan Külekci for his guidance, patience and valuable suggestions in thesis research and scientific publication. It has been an enormous pleasure and privilege to have worked with him.

I'm truly grateful for the continuous technical support of UHEM staff about clusters systems and tools while running and profiling program. Also, I would like to thank co-worker Semih Tunacı for usage of his laboratory workstation when developing the initial programs on the Intel Xeon processor.

A special thanks to my family for their support and motivation during academic studies and whole my life.

December 2018 Mehmet Akif AYDOĞMUŞ

## TABLE OF CONTENTS

## ABBREVIATIONS

| | |
|---|---|
| **AVX** | : Advanced Vector Extensions |
| **CPI** | : Cycle Per Instructions |
| **CPU** | : Central Processing Unit |
| **EBS** | : Event-Based Sampling |
| **EPSM** | : Exact Packed String Matching |
| **GCC** | : GNU Compiler Collection |
| **GPU** | : Graphical Processing Unit |
| **HPC** | : High-Performance Computing |
| **Patlen** | : Pattern Length |
| **L1, L2** | : Level-1, Level-2 |
| **SAD** | : Sum of Absolute Difference |
| **SSE** | : Streaming SIMD Extensions |
| **SSEF** | : Streaming SIMD Extensions Filtering |
| **UHeM** | : Ulusal Yüksek Başarımlı Hesaplama Merkezi (National HPC Center) |
| **OS** | : Operating System |

## SYMBOLS

| | |
|---|---|
| Σ | **:** Alphabet Size |
| **ms** | **:** Mili Second |
| **KB** | **:** Kilo Byte |
| **MB** | **:** Mega Byte |
| **Hz** | **:** Hertz |
| *P* | **:** Pattern |
| *T* | **:** Text |
| **m** | **:** pattern length |
| **n** | **:** text size |

## LIST OF TABLES

## LIST OF FIGURES

## OPTIMIZING PACKED STRING MATCHING On AVX2 PLATFORM

## SUMMARY

Exact string matching, searching for all occurrences of given pattern $P$ on a text $T$, is a fundamental issue in computer science with many applications in natural language processing, speech processing, computational biology, information retrieval, intrusion detection systems, data compression, time series analysis and etc. The speed of string matching is gaining more importance due to today applications of scientific and industrial are operating on large datasets increasing continuously. Accelerating the pattern matching operations benefiting from the SIMD parallelism has received attention in the recent literature, where the empirical results on previous studies revealed that SIMD parallelism significantly helps, while the performance may even be expected to get automatically enhanced with the ever increasing size of the SIMD registers.

The variants of the previously presented EPSM and SSEF algorithms are proposed, which are originally implemented on Intel SSE4.2 (Streaming SIMD Extensions 4.2 version with 128-bit registers). The new algorithms are designed according to Intel AVX2 platform (Advanced Vector Extensions 2 with 256-bit registers) and the gain in performance is analyzed with respect to the increasing length of the SIMD registers. Particularly in algorithms based on filtering methods, memory access time can be the performance bottleneck when working with large decimal values such as 32-bit filter values calculated from 256-bit registers. Profiling the new algorithms by using the Intel Vtune Amplifier for detecting performance issues led us to consider the cache friendliness in the AVX2 platform. Hardware Event-Based Sampling (EBS) analysis type is selected for profiling on various filter lengths and data structures and so performance metrics of algorithms are collected. Cache optimization techniques are applied to overcome the problems particularly addressing the search algorithms based on filtering.

Experimental comparison of the new solutions with the previously known-to-be-fast algorithms on small (genome sequence), medium (protein sequence), and large alphabet (English language) text files with diverse pattern lengths showed that the algorithms on AVX2 platform optimized cache obliviously outperforms the previous solutions. It can be inferred from experiments that proposed algorithms are practicable for today applications. Also, proposed algorithms can be portable to other architectures like the ARM and AMD using equivalent SIMD instructions with some modifications.

# AVX2 PLATFORMU ÜZERİNDE PAKETLENMİŞ DİZGİ EŞLEŞTİRME VE OPTİMİZASYONU

## ÖZET

Dizgi eşleştirme verilen bir örüntünün $P$ bir metin $T$ üzerinde bir veya tüm eşleşmelerini bulma işlemi olarak tanımlanabilir ve bilgisayar bilimlerinde üzerinde geniş bir şekilde çalışma yapılan temel bir konudur. Dizgi eşleştirmenin birçok alanda uygulaması vardır, bunlar arasında doğal dil işleme, ses işleme, hesaplamalı biyoloji, bilgi gerigetirimi, saldırı tespit sistemleri, arama motorları, veri sıkıştırma, zaman serileri analizi gibi konular sayılabilir. Günümüzde farklı alanlardaki verilerin hızı, kapasitesi ve çeşitliliği sürekli artmaktadır, dolayısıyla bu büyük veri yığınları üzerindeki dizgi eşleştirme algoritmalarının hızları gittikçe önem kazanmaktadır. Özellikle zaman-kritik uygulamalarda dizgi eşleştirme algoritmalarının performansı en önemli kriterlerden biridir. Literatürde dizgi eşleştirme problemin tipine göre kategorilere ayrılmıştır; tam, yaklaşık, dairesel, karmaşık, sıra-korumalı olarak eşleştirme tipleri mevcuttur.

Bu çalışmada tam dizgi eşleştirme için paketleme metoduna dayalı yeni algoritmalar geliştirilmiştir. Tam dizgi eşleştirme bir metin üzerinde verilen örüntünün tam eşleşen alt dizgilerini bulma olarak tanımlanmaktadır. Bu tezde yeni önerilen paketlenmiş dizgi eşleştirme algoritmalarında SIMD (Single Instruction Multiple Data) teknolojisinden yararlanılmıştır. SIMD teknolojisi bir komutla aynı anda çoklu veri üzerinde işlem yapma olanağı sunmaktadır. Ayrıca geliştirilen algoritmalarda optimum çözüme ulaşmak için algoritmalar üzerinde analiz yapılarak optimizasyonlar uygulanmıştır.

1970 yılından günümüze çok sayıda dizgi eşleştirme algoritmaları sunulmuştur, günümüzde ise daha iyi sonuçlara ulaşabilmek amacıyla yeni bakış açılarıyla yöntemler geliştirilmeye devam edilmektedir. SIMD ile paralleştirmeye dayalı yöntemlerle dizgi eşleştirme operasyonunu hızlandırma son yıllarda literatürde görülmeye başlanmıştır. Yapılan deneysel çalışmalarda SIMD ile veri seviyesinde paralelleştirme önemli derecede hızlanma sağlamaktadır, ayrıca artan "register" boyutlarıyla bu performans artışının devam etmesi beklenmektedir. Bu çalışmada daha önce 128 bit "register" boyutuna sahip "Intel SSE4.2 (Streaming SIMD Extensions 4.2)" teknolojisi üzerinde geliştirilen EPSM ve SSEF algoritmalarının yeni optimal varyantları önerilmiştir. Yeni önerilen algoritmalar 256 bit "register" boyutuna sahip Intel AVX2 (Advanced Vector Extensions 2) platformu üzerinde geliştirilmiştir ve artan "register" boyutuna karşılık performansın nasıl arttığı ve hangi sorunlarla karşılaşıldığı analiz edilmiştir. Bu bağlamda Intel Vtune Amplifier uygulaması ile algoritmalar üzerinde profilleme ile performans darboğazları tespit edilerek sebepleri araştırılmış ve bu bilgiler ışığında algoritmalar düzenlenmiştir.

Çalışmada öncelikle AVX2 "intrinsic" fonksiyonları kullanılarak 256-bit veri üzerinde sabit zamanda çalışan kelime-boyu komutlar tanımlanmıştır. Daha sonra bu

kelime-boyu komutlar kullanılarak yeni algoritmalar geliştirilmiştir. AVX2 platformu üzerinde çalışan bu komutlardan bahsedecek olursak; "kelime-boyu karşılaştırma komutu" argüman olarak aldığı iki adet 256-bit veriyi karşılaştırarak eşleştirme sonuçlarını 32 bit veri biçiminde vermektedir. "Kelime-boyu eşleştirme komutu" ise 256 bit iki veride dörtlü karakter grupları üzerinde 16 adet Mutlak Farklar Toplamı (SAD) hesaplayarak eşleşme olan durumları belirlemekte ve eşleşme sonuçlarını 32-bit veri olarak döndürmektedir. "Kelime-boyu değiştirme komutu" iki adet 256 bit veri üzerinde 32 karakterlik bölütlerin yerlerini değiştirerek istenen veri düzenini sağlamakta, bu yer değiştirme işlemini argüman olarak aldığı istenen sıralamayı temsil eden değişkene göre uygulamaktadır. Tanımlanan diğer "kelime-boyu filtre hesaplama komutu" 256 bitlik veri üzerinde kaydırma işlemi yaptıktan sonra tüm baytların en anlamlı bitlerini kullanarak 32 bitlik fitre değerini üretir. Kaydırma işlemindeki kaydırma miktarı üzerinde işlem yapılan verinin dahil olduğu alfabeye göre belirlenen 0 ile 7 arasında bir değerdir.

Kelime-boyu komutlar kullanılarak EPSMA (EPSMA-1,2,3) ve SSEFA algoritmaları önerilmiştir. ESPMA-1 algoritması örüntü boyunun 8'den küçük olduğu durumlar için geliştirilmiştir ve temeli "kelime-boyu karşılaştırma komutu" kullanımına dayanmaktadır. Örüntü boyunun 8 ile 32 arasında olduğu durumlar için "kelime-boyu eşleştirme komutu" baz alınarak EPSMA-2 algoritması tasarlanmıştır. Ayrıca EPSMA-2 algoritmasında ardışıl eşleştirme komutları arasında "kelime-boyu değiştirme komutu" verinin düzenlenmesi amacıyla kullanılmıştır. EPSMA-3 algoritması ise "kelime-boyu filtre hesaplama komutu" ile filtreleme metoduna dayanmaktadır ve boyu 32 ile 64 arasında olan örüntüler için uygundur. 16 karakterlik bölütü temsil eden filtre 16 bitten oluşmaktadır ve bir filtre hesaplama komutuyla aynı anda 16 bitlik 2 adet filtre elde edilmektedir. EPSMA-3'de filtreler ayrı ayrı uygulanarak olası eşleşme adayları elde edilir ve bu adaylar üzerinde tam eşleşme karşılaştırması yapılır. Boyu 64'den büyük olan uzun örüntüler için ise SSEFA algoritması önerilmiştir ve bu algoritma da "kelime-boyu karşılaştırma komutu" tabanlı filtreleme yaklaşımına dayanmaktadır. Ancak SSEFA'da filtreler 32 karakter bölütü baz alınarak hesaplandığından her 32 karakteri 32 bitten oluşan bir filtre değeri temsil eder. 32-bit filtre değeri indeks olarak kabul edilip filtre vektörü oluşturulduğunda ise vektörün boyutu filtre değerinin alabileceği maksimum değer ile belirlenir. 32 bit filtre değeri kullanıldığında vektörün boyutu 4GB olacaktır, metin üzerinde her filtre arama işleminde vektörün ilgili elemanına erişim gerekeceğinden hafızaya erişimdeki gecikmeler performans üzerinde önemli bir performans kısıtı oluşturacaktır.

Optimal filtre uzunluğu ve veri tipini belirlemek için Intel Vtune Amplifier aracı ile performans analizi yapılmıştır. Vtune Amplifier Intel tarafından geliştirilen yüksek performanslı hesaplama alanında modern mimariler üzerinde performans analizi yaparak olası sorunların kaynağını tespit eden ve daha hızlı uygulamalar için yol gösteren bir analiz aracıdır. Hafızaya erişim analizi için "Hardware Event-Based Sampling (EBS)" analiz tipi seçilerek farklı filtre uzunlukları ve veri tipleri için geçen süreler, komut başına çevrim sayısı (CPI) ve L1 kayıp oranı (miss rate) değerleri elde edilmiştir. Yapılan analizler sonucu hafızaya erişmede önbelleğin optimal kullanımı açısından en uygun filtrenin 14 bit uzunluğunda olduğu ve filtrelerin bağlı liste yarine array yapısında tutulması gerektiği tespit edilmiştir. Arama işleminde filtrelemeden geçebilen eşleşme adayı sayısını azaltmak için 32 bit ham filtrenin her bir yarısındaki ayrık bitlerden oluşan 2 adet 14 bit filtre ardışıl olarak uygulanmıştır.

Algoritmalar C programlama dilinde yazılmıştır, performans testleri ise dizgi eşleştirme için test platformu sunan ve literatürdeki bilinen tüm algoritmaları içeren SMART ile yapılmıştır. Performans karşılaştırma işlemi Σ alfabe boyutu 4 olan genom sekansı, 20 olan protein sekansı ve 128 olan ingilizce metin olmak üzere 3 farklı verisetinde ve literatürde en hızlı olarak bilinen diğer algoritmalar teste eklenerek farklı örüntü uzunlukları üzerinde yapılmıştır. Performans testleri tam dizgi eşleştirme problemi için yeni önerilen algortimaların önceki en verimli olarak kabul edilen algoritmalardan farklı örüntü uzunlukları ve alfabe boyları seçeneklerinin %90'nından fazlasında daha iyi sonuç verdiğini göstermiştir. Sonuç olarak geliştirilen algoritmaların farklı alfabelerde ve örüntü boylarında pratik uygulamalar için son derece kullanışlı olduğu söylenebilir. Gelecekte ise önerilen algoritmalar ARM, AMD gibi diğer işlemci mimarilerindeki eşdeğer SIMD fonksiyonlarıyla modifiye edilerek o mimarilerde de çalışabilecek hale getirilip algoritmaları kullanan uygulamaların taşınabilirliği arttırılabilir.

# 1. INTRODUCTION

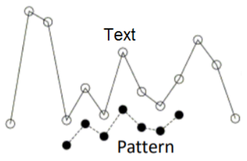
## 1.1 String Matching

String matching, finding one or all occurrences of given string (also called pattern) on a text, is a fundamental and widely studied issue in Computer Science. As a subdomain of text processing, string matching gains more importance with the progress and spread technology. Particularly, volume, variety and velocity of data are increasing day to day on different areas thus the speed of string matching is gaining importance. String matching has many applications in diverse fields such as such as natural language processing, information retrieval, data compression, computational biology and chemistry, intrusion detection systems, image and signal processing, speech processing, time series analysis. Due to its wide usage in various applications, string matching is a subject that continues to be studied. In literature, there are various types of string matching as exact, approximate, circular, jumbled, order-preserving matches. These matching types are described below.

- Exact Matching: Finding all conditions where pattern P occurs exactly same as a substring of text.
- Approximate Matching: Finding all approximate occurrences of the pattern in the text with finite number chraracters.
- Circular Matching: Finding all occurrences the circular rotations of a pattern in a text.
- Jumbled Matching: Finding all permuted occurrences of a pattern in a text.
- Order-Preserving Matching: Finding all the substrings which have the same length and relative order (numerical order of numbers in a string) as the pattern in a text.

On the other hand, string matching algorithms are classified into 4 classes in terms of method by Faro and Lecroq [1]:

**Table 1.1** : Types of string matching.

| Sample Text: | the quick **brown fox** jumps |
|---|---|
| Exact Match: | brown fox |
| Approximate Match: | grown fix |
| Circular Match: | own foxbr |
| Jumbled Match: | wonb xrof |
| Order-Preserving Match: |  |

1. Comparison Algorithms: apply comparisons between characters.

2. Automata Algorithms: make use of deterministic automata.

3. Bit-parallelism Algorithms: simulate the behavior of non-deterministic automata.

4. Packed Algorithms: multiple characters are packed and comparing are performed in bulk.

## 1.2 Purpose of Thesis

In this work new efficient algorithms are developed using packed method for exact string matching which has more usage areas according to the other types of string matching. Processing time of string matching is crucial for today applications and scientific researches where the large amount of data are stored and streamed. SIMD (Single Instruction Multiple Data) technology is used for packed string matching and then cache optimization is applied with profiling to achieve optimum solution.

## 1.3 Literature Review

Numerous string matching algorithms have been presented since the 1970s with the various theoretical point of view. Even so, new methods are still being developed to achieve better searching times. Knuth-Morris-Pratt algorithm is based on finite automata with $\mathcal{O}(m)$ and $\mathcal{O}(n)$ time complexity [2]. Boyer-Moore algorithm is a combination of heuristic approaches which are "bad character heuristic" and "good suffix heuristic" with $\mathcal{O}(mn)$ worst-case time complexity (best case time $\mathcal{O}(n/m)$) [3]. Backward-DAWG- Matching algorithm (BDM) [4] is based on the suffix automaton for the reversed pattern and it has asymptotic optimum average time complexity $\mathcal{O}(n(\log_\sigma m)/m)$ especially for the long pattern.

Filter based solutions have been also developed for string matching, Karp-Rabin algorithm [5] is the first filtering algorithm using a hashing function with $\mathcal{O}(mn)$ time complexity but $\mathcal{O}(n\ m)$ expected running time. Another filtering algorithm is Q-Gram (QF) which is based on consecutive q-grams in the text with $\mathcal{O}(mn)$ worst case complexity and $\mathcal{O}(nq/(m-q))$ best case complexity.

An extensive review of the string matching algorithms between 2000-2010 and comprehensive experimental evaluation of 85 exact string matching algorithms are presented by Faro and Lecroq, [6] [7] respectively.

In recent years, the usage of SIMD (Single Instruction Multiple Data) instructions in string matching algorithms is appeared [8–12]. Tarhio et al. [13] newly proposed the algorithm compares 16 or 32 characters in parallel by applying SSE2 and AVX2 instructions besides they use the increasing order for comparisons of pattern symbols to achieve better results. In this work, Intel-AVX2 (Advanced Vector Extensions 2) based variations of the EPSM [8] and SSEF [9] algorithms are proposed from a different point of view. Especially, practical efficiency of new algorithms is focused on while developing algorithms and so the algorithms are optimized in this manner according to the profiling results.

## 1.4 Thesis Structure

The sections of the thesis are arranged as follows; firstly notions of exact string matching, SIMD technology and used Intel AVX2 intrinsics are explained in chapter 2 named as BASICS. In chapter 3 PROPOSED ALGORITHMS, word-size instructions which are the main component of algorithms are discussed in details. Then EPSMA (EPSMA-1,2,3) and SSEFA algorithms are expressed, also profiling and cache optimization of algorithms are given as the next topic of chapter 3. Experimental results of performance comparisons are presented in tables and figures for various dataset types and pattern lengths in chapter 4 EXPERIMENTAL RESULTS. Finally, results are evaluated and potential future works are given in chapter 5 called as CONCLUSION AND RECOMMENDATIONS.

## 2. BASICS

In this section, firstly terminology of exact string matching is given, after that Intel SIMD and AVX2 technology is explained in detail with related intrinsics which will be used in new algorithms.

### 2.1 Notions of Exact String Matching

The exact string matching problem is described as counting all the occurrences of a pattern $P$ of length m in a text $T$ of length n assuming m $\ll$ n, over a finite alphabet $\Sigma$.

String p of length $m$ 0 can be defined as character array $p[0...m-1]$ over the finite alphabet $\Sigma$ of size $\sigma$ and $p[i]$ corresponds the $(i + 1)$-st character for $0 \leq i$ $m$. Substring of p between indexes $(i + 1)$-st and the $(j + 1)$-st characters is represented by $p[i...j]$ while $0 \leq i \leq j$ $m$. Also it can be expressed as $p_i$ $p[i]$ and $p$ $p_0 p_1 ... p_{m-1}$. Using notations above, exact string matching is searching of condition as $p_0 p_1 ... p_{m-1}$ = $t_i t_{i1} ... t_{im-1}$ where the text is $T$ $t_0 t_1 ... t_n$. Bitwise operators are employed on computer words in the algorithms such as bitwise AND "&", bitwise OR "|" and left shift "$\ll$" where computer word size is denoted by $w$.

A string can be represented as $S$ $s_0 s_1 ... s_{k-1}$ where k is the number of the characters and each character corresponds to the single byte. The bits of single byte $s_i$ can be defined as bit array like $s_i$ $b^i_0 b^i_1 b^i_2 b^i_3 b^i_4 b^i_5 b^i_6 b^i_7$ where $b^i_0$ is the msb bit. The chunk of 32-byte is represented as $C^i$ $s_{32 \cdot i} s_{32 \cdot i1} s_{32 \cdot i2} ... s_{32 \cdot i31}$ so string is described in terms of 32-byte chunks as $S$ $C^0 C^1 C^2 ... C^z$ where z = $\lfloor (k - 1)/32 \rfloor$ and $0 \leq i \leq$ z. If $k$ value is not divisible by 32, the last chunk $C^z$ is incomplete and zero padding is applied for the rightmost empty part as $s_l$ =0 where $k$-1<$l$. The number of 32-byte chunks of text ($T$) and pattern ($P$) are shown as $N = \lceil n/32 \rceil$ and $M = \lceil m/32 \rceil$. The chunk and byte symbols of text ($T$) and pattern ($P$) are presented such as:

- Text Representation: single byte $t_i$, $0 \leq i <$ n; 32-byte chunks: $Y^i$, $0 \leq i <$ N
- Pattern Representation: single byte $p_i$, $0 \leq i <$ m; 32-byte chunks: $R^i$, $0 \leq i <$ M

| Chunks: | $Y^0$ | $Y^1$ | ... | $Y^{N-1}$ |
|---|---|---|---|---|
| Bytes: | $t_0t_1...t_{31}$ | $t_{32}t_{33}...t_{63}$ | ... | $t_{32\cdot(N-1)}t_{32\cdot(N-1)1}\cdots t_{n-1}$ |

| Chunks: | $R^0$ | $R^1$ | ... | $R^{M-1}$ |
|---|---|---|---|---|
| Bytes: | $p_0p_1\cdots p_{31}$ | $p_{32}p_{33}\cdots p_{63}$ | ... | $p_{32\cdot(M\ \ 1)}p_{32\cdot(M\ \ 1)1}\cdots p_{m\ \ 1}$ |

## 2.2 SIMD

SIMD (Single Instruction Multiple Data) technology allows one instruction can be operated at the same time on multiple data items. In 1996, Michael J. Flynn classified the computer architectures according to the number of instruction and data stream into four major categories as SISD, SIMD, MISD and MIMD [14]. This classification becomes a reference tool for designing of modern processors. On the other hand modern microprocessor architectures may have more than one of defined classification type above.

- SISD: Single instruction operates on single data element
  e.g.: Traditional von Neumann single CPU computer

- SIMD: Single instruction operates on multiple data elements
  e.g.: Array processor, Vector processor

- MISD: Multiple instructions operate on single data element
  e.g.: Fault-tolerant computers, Near memory computing (Micron Automata processor).

- MIMD: Multiple instructions operate on multiple data:
  e.g.: Multiprocessor, Multithreaded processor

The following figure depicts the high-level computer architectures in terms of Flynn's classifications. In the diagrams, PU corresponds the "Processing Unit" that performs the instruction.

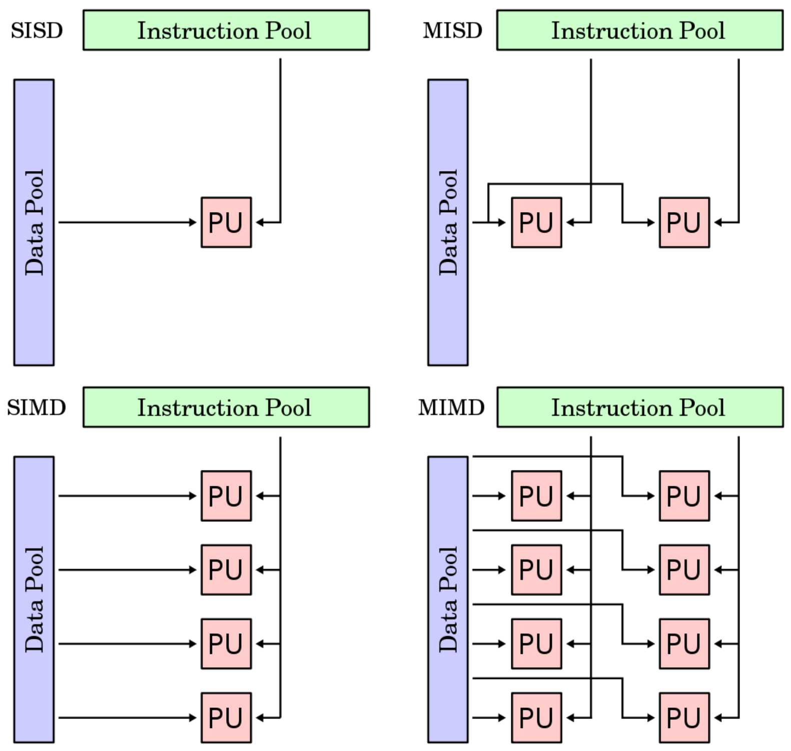


**Figure 2.1** : Flynn's classification.

Also Parallelism can be categorized in terms of application as TLP, DLP, and ILP.

- Task Level (Thread-level) Parallelism (TLP): Multiple processes/tasks/threads sequences of the same application can be executed simultaneously. However, it may has some bottlenecks in practice like communication/synchronization overheads according to the algorithm characteristics.
- Data Level Parallelism (DLP): One Instruction can be executed concurrently on multiple data streams such as SIMD parallelism. Non-regular data access pattern and memory bandwidth can be significant issues in total performance.
- Instruction Level Parallelism (ILP): Several independent instructions of program can be operated in parallel (overlapping instructions) by a processor. ILP, also callled Fine-grained parallelism, is constrained by the potential data and control dependencies.

SIMD technology was first used as the vector processor of ILLIAC IV in the 1966, it became as the basis for vector supercomputer of Cray, CDC Star-100 and Texas Instruments ASC in the early 1970s. Afterwards, vector processor was defined separately from SIMD processor because of the time-space duality, therefore SIMD operation was accepted as a array processing [15]. On the other hand, modern computer architectures combine array and vector processing by applying data parallelism in both time and space.

*Time-space Duality:*

- Array processor: Instruction performs the operation on multiple data items at the same time using different spaces.
- Vector processor: Instruction performs the operation on multiple data items in consecutive time steps using the same space.

At first, SIMD technology is developed for multimedia purposes such as image and audio file processing, moreover it has been used in scientific researches in course of time like cryptography,text and data processing. Besides super-computers, microprocessor vendors also supply SIMD processing for workstation and desktop-computer due to widespread of SIMD usage in the applications. Therefore modern ISAs (Instruction Set Architectures) include SIMD operations, for instance; Intel MMX/SSEn/AVX, PowerPC(IBM) AltiVec, ARM Advanced SIMD etc.

## 2.3 Intel SIMD and AVX2

In 1996, Intel introduced SIMD as the MMX (MultiMedia eXtension) technology with Pentium processor which designed especially for improved performance on multi-media applications. After that, Intel released SSE (Streaming SIMD Extensions) and INTEL AVX (Advanced Vector Extensions) in order to provide more acceleration with SIMD techolongy operated on wider register lengths. Intel Xeon Processor which has AVX2 instructions and floating point fused multiply-add (FMA) instructions is used for new algorithms in this work.

Intel AVX2 (Advanced Vector Extensions) includes various intrinsics operated on 256-bit register data, providing enhanced functionality for broadcast/permute operations, vector shifting/permutation operations and fetch instructions for non-contiguous data elements from memory. AVX2 architecture consists of the 16 256-bit YMM registers called YMM0-YMM15 and 32-bit control/status register called MXCSR [16].

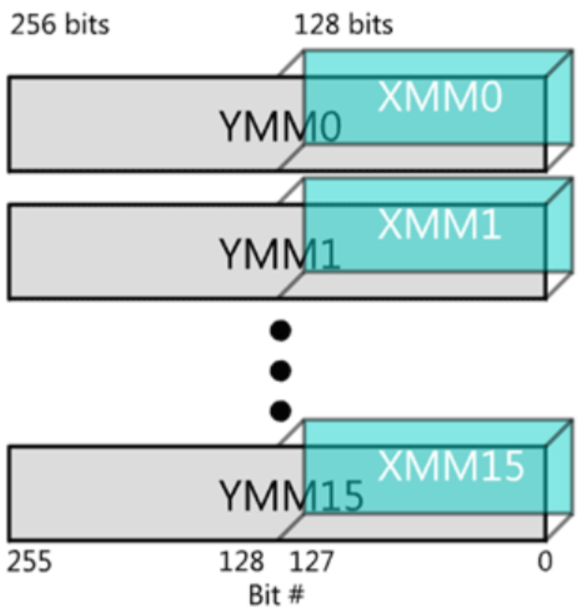


**Figure 2.2 :** YMM registers share bits with the XMM registers.

By the way 128 less significant bits of YMM registers are overlapped with the older 128-bit XMM registers used for Intel SSE as shown in figure 2.2

## 2.4 Used AVX2 Intrinsics For New Algorithms

Operation of instructions are sketched using the "Intel 64 and IA-32 Architectures Software Developer's Manual" [17].

- **_mm256_setr_epi32** :

  *prototype*: __m256i _mm256_setr_epi32 (int e7, int e6, int e5, int e4, int e3, int e2, int e1, int e0)

  *description*: Intrinsic sets packed 32-bit integer with the given values in reverse order and stores the result in 256-bit data.

- **_mm256_movemask_epi8**:

  *prototype*: int _mm256_movemask_epi8 (__m256i a)

  *description*: Intrinsic creates mask from the most significant bit of each 8-bit element in *a*, and store the result in 32-bit data.

  *instruction*: `vpmovmskb` r32, ymm

- **_mm_popcnt_u32**:

  *prototype*: int _mm_popcnt_u32 (unsigned int a) *description*: Intrinsic counts the number of bits set to 1 in unsigned 32-bit integer *a*, and return that count in 32-bit data.

*instruction*: `popcnt` r32, r32

- **_mm256_cmpeq_epi8** :

  *prototype*: __m256i _mm256_cmpeq_epi8 (__m256i a, __m256i b)

  *description*: Intrinsic compares packed 8-bit integers in 256-bit a and 256-bit b for equality, and returns the result as 256-bit data.

  *instruction*: `vpcmpeqb` ymm, ymm, ymm

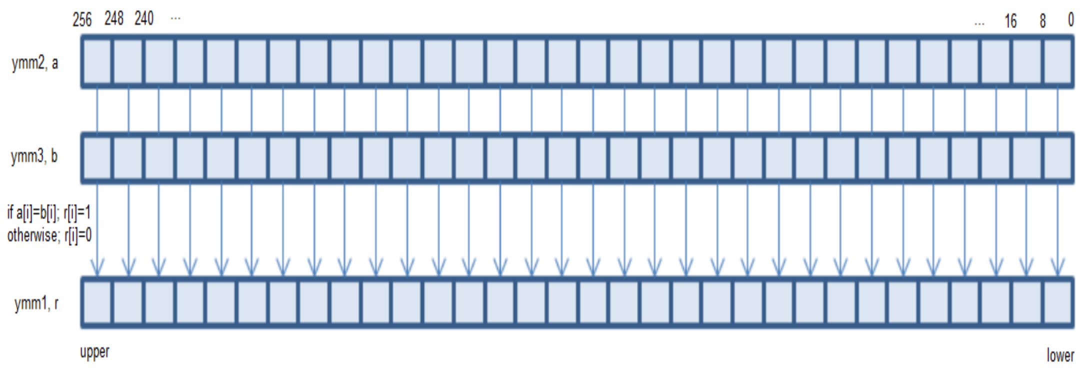


**Figure 2.3** : Sketch of VPCMPEQB operation.

- **_mm256_mpsadbw_epu8**:

  *prototype*: __m256i _mm256_mpsadbw_epu8 (__m256i src1, __m256i src2, const int imm8)

  *description*: Intrinsic computes the multiple packed sums of absolute difference (SAD) between given 4-byte sub-vector from *src*2 data and eight subsequent 4-byte sub-vector from *src*1 data. *imm*8 variable is the offset and specifies the starting point of quadruplets on *src*2. 8 packed SAD values are calculated on each 128-bit lanes of 256-bit data (src2) separately as depicted in figure 2.4, eventually 16 packed SAD values are obtained in total.

  *instruction*: `vmpsadbw` ymm, ymm, ymm, imm

- **_mm256_permute2f128_si256** :

  *prototype*: __m256i _mm256_permute2f128_si256 (__m256 a, __m256 b, int imm8)

  *description*: Intrinsic shuffles 128-bit fields selected by imm8 from a and b, and returns the result of permute operation in 256-bit data. The source for the first destination 128-bit field is selected by imm8[1:0] and the source for the second destination field is selected by imm8[5:4] as shown in figure 2.5.

*instruction*: `vperm2f128` ymm, ymm, ymm, imm

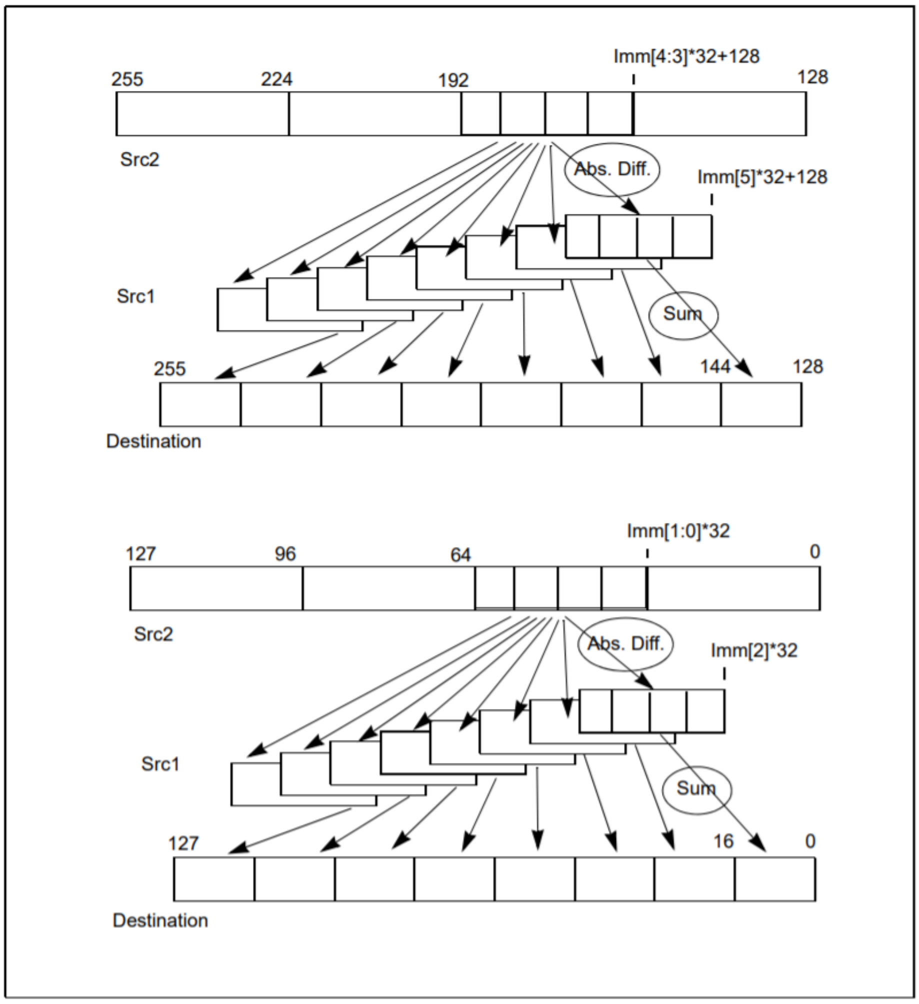


**Figure 2.4** : Sketch of VMPSADBW instruction.

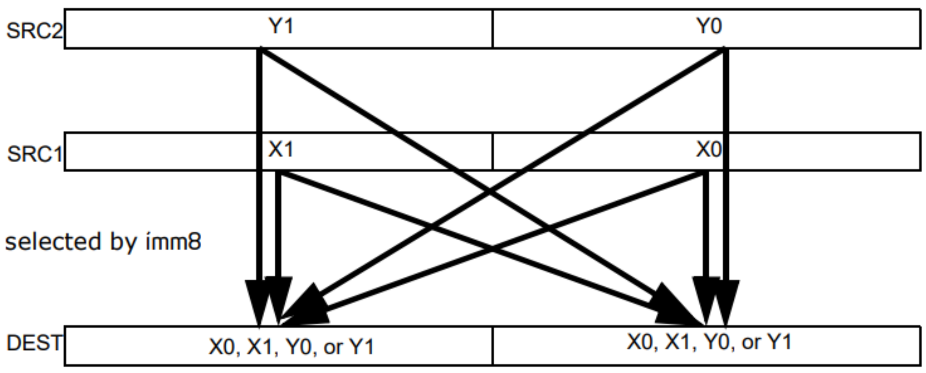


**Figure 2.5** : Sketch of VPERM2F128 operation.

- **_mm256_permutevar8x32_epi32** :

  *prototype*: __m256i _mm256_permutevar8x32_epi32(__m256i a, __m256i idx)

  *description*: Intrinsic shuffles 32-bit integers between across lanes using the input permute variable idx and stores the result in 256-bit data as depicted in figure 2.6.

  *instruction*: `vpermd` ymm, ymm, ymm

  Latency:1, Throughput:1 (for Intel Broadwell architecture)

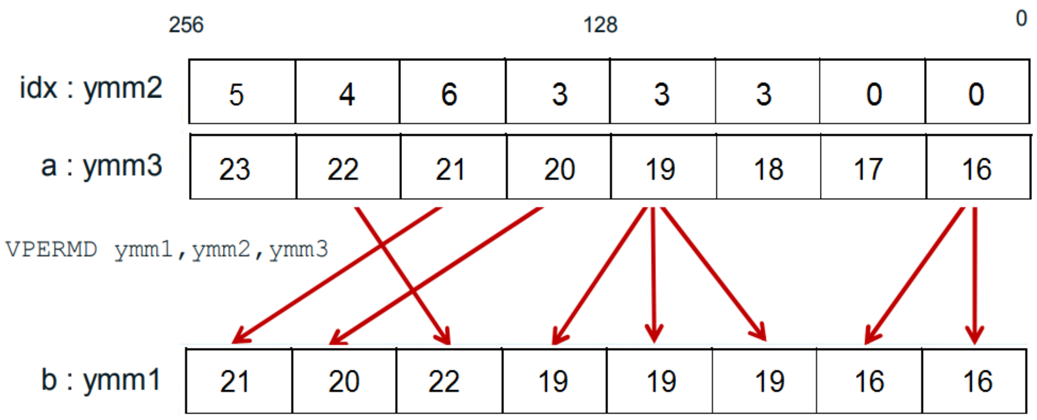


**Figure 2.6** : Sketch of VPERMD operation

- **_mm256_slli_epi64** :

  *prototype*: __m256i _mm256_slli_epi64(__m256i a, int imm8)

  *description*: Intrinsic left shifts packed 64-bit integers by input imm8 while padding zeros and stores the result in 256-bit data as shown in figure 2.7.

  *instruction*: `vpsllq` ymm, ymm, imm

  Latency:1, Throughput:1 (for Intel Broadwell architecture)

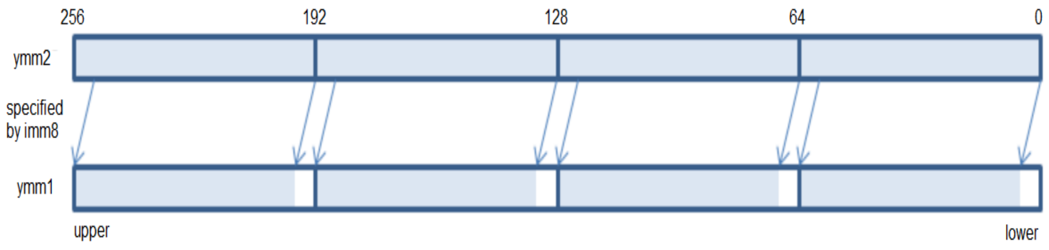


**Figure 2.7** : Sketch of VPSLLQ operation.

# 3. PROPOSED ALGORITHMS

Initially in this chapter, descriptions of word-size instructions which are composed of AVX2 intrinsics are given, after that newly proposed exact matching algorithms based on word-size instructions are presented in detail.

## 3.1 Word-Size Instructions

Specialized word-size instruction could be emulated in constant time by using AVX2 intrinsics previously described in section 2.4.

### 3.1.1 wscmp_a(a, b) (word-size compare instruction on AVX2)

Before, wscmp instruction is defined as; $a = a_0a_1...a_{\alpha-1}$ and $b = b_0b_1...b_{k-1}$, wscmp returns an $\alpha$ bit value, $r = r_0r_1...r_{\alpha-1}$ where $r_i = 1$ if and only if $a_i$=$b_i$, $r_i$=0 otherwise. wscmp_a operation can be emulated with 256-bit SIMD intrinsics instead of using 128-bit intrinsics as used in EPSM algorithm.

```
h ← _mm256_cmpeq_epi8(a,b)
r ← _mm256_movemask_epi8(h)
```

| Char: | 1 | 2 | 3 | 4 | 5 | ...... | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|
| a: | 01110100 | 01100011 | 01100111 | 01100001 | 01100011 |  | 01100001 | 01110100 | 01100111 |
| b: | 01100011 | 01100001 | 01100111 | 01110100 | 01100011 |  | 01100011 | 01110100 | 01100001 |
|  |  | ⇓ |  |  | wscmp_a(a, b) |  |  | ⇓ |  |
| r: | 0 | 0 | 1 | 0 | 1 |  | 0 | 1 | 0 |

**Figure 3.1** : Example operation of word-size compare instruction.

_mm256_cmpeq_epi8 instruction compares packed 8-bit integers in 256-bit a and 256-bit b for equality, and returns the result as 256-bit data. _mm256_movemask_epi8 instruction creates mask using the most significant bit of each 8-bit element in 256-bit $h$ and returns the 32-bit result as $r$ . The diagram 3.1 shows an example of the wscmp_a(a; b) operation when working with characters in ASCII code on 256-bit registers, assuming $a$ and $b$ variables are composed of 32 characters such as a:"*tcgac...atg*" and b:"*cagtc...cta*".

### 3.1.2 wsmatch_a(a, b) (word-size matching instruction on AVX2 )

wsmatch instruction is defined as; $a = a_0a_1...a_{\alpha-1}$ and $b = b_0b_1...b_{\alpha-1}$, wsmatch returns an $\alpha$ bit integer value, $r = r_0r_1...r_{\alpha-1}$ where $r_i = 1$ if and only if $a_{ij} = b_j$ for $j = 0...k-1$ [8]. Also, let z be a 256-bit register with all elements are set to zero by `_mm256_setzero_si256` intrinsic. Operation of wsmatch instruction is emulated with 256-bit SIMD intrinsics named as wsmatch_a instead of using 128-bit intrinsics.

$h \leftarrow$ `_mm256_mpsadbw_epu8` $(a,b,imm8)$

$h \leftarrow$ `_mm256_cmpeq_epi8` $(h,z)$

$r \leftarrow$ `_mm256_movemask_epi8` $(h)$

`_mm256_mpsadbw_epu8` intrinsic calculates 16 SADs (Sum of Absolute Difference) in total, however, operates on 128-bit lanes separately instead of the whole 256-bit data. This condition requires additional shuffling of input data in order to properly arrange the data bytes compatible with instruction. Meanwhile, the following wspermute_a intrinsic can used to make this shuffling. The diagram 3.2 includes an example of the wsmatch_a(a; b) operating with characters in ASCII code on 256-bit registers, assuming a:"*gatcatgct...*" (32 characters) and b:"*tcat*" (4 characters).

| Char: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... | 32 |
|---|---|---|---|---|---|---|---|---|---|---|
| a: | 01100111 | 01100001 | 01110100 | 01100011 | 01100001 | 01110100 | 01100111 | 01100011 | ... | ... |
| b: | 01110100 | 01100011 | 01100001 | 01110100 | | | | | | |
| | | ⇓ | | `wsmatch_a(a, b)` | | ⇓ | | | | |
| r: | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | ... | ... |

**Figure 3.2** : Example operation of word-size match instruction.

As shown in the above example 3.2, characters of $a[3:6]$ and $b[1:4]$ are same as "*tcat*" (in decimal:"116-99-97-116"; in binary:"01110100-01100011-01100001-01110100"). Therefore, SAD value will be zero between these quadruplets, $a[3:6]$ and $b[1:4]$. In result, $r[3]$ value will be 1 after applying _mm256_cmpeq_epi8 intrinsic with zero array $z$ and masking operation by _mm256_movemask_epi8.

### 3.1.3 wspermute_a(a,b) (word-size permute instruction on AVX2 )

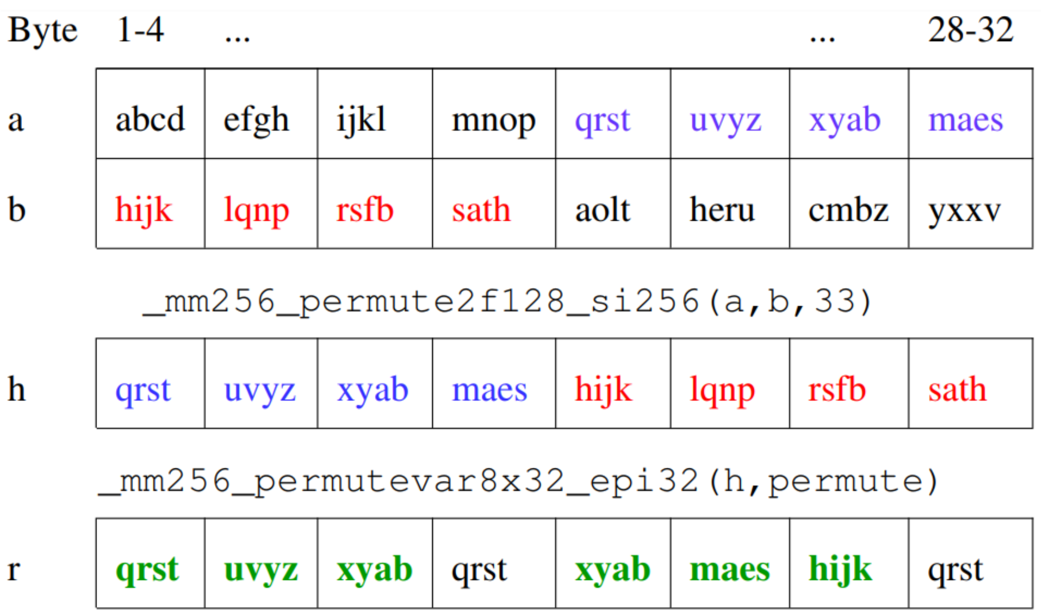
wspermute_a instruction corresponds to the wsblend instruction of EPSM algorithm and it arranges the 256-bit input data in the right order required for SADs operation.

- imm8 : index variable for permute function

```
h = _mm256_permute2f128_si256 (a, b, imm8);
permute = _mm256_setr_epi32(0, 1, 2, 0, 2, 3, 4, 0);
r = _mm256_permutevar8x32_epi32(h, permute);
```

_mm256_permute2f128_si256 intrinsic shuffles 128-bit data lanes of 256-bit *a* and *b* according to the *imm*8 and return the arranged data in 256-bit *h*. _mm256_setr_epi32 intrinsic assigns eight input data with the packed 32-bit integer to 256-bit data in reverse order and returns the created 256-bit data as *permute*. Finally, the _mm256_permutevar8x32_epi32 shuffles 32-bit integers between across lanes using the input *permute* variable and returns the 256-bit result named as *r*. An example operation of wspermute_a instruction is given in the figure 3.3.

*imm*8 variable : 33, *permute* variable: "0, 1, 2, 0, 2, 3, 4, 0".



| Byte | 1-4 | ... | | | | | ... | 28-32 |
|---|---|---|---|---|---|---|---|---|
| a | abcd | efgh | ijkl | mnop | qrst | uvyz | xyab | maes |
| b | hijk | lqnp | rsfb | sath | aolt | heru | cmbz | yxxv |

_mm256_permute2f128_si256(a,b,33)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| h | qrst | uvyz | xyab | maes | hijk | lqnp | rsfb | sath |

_mm256_permutevar8x32_epi32(h,permute)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| r | **qrst** | **uvyz** | **xyab** | qrst | **xyab** | **maes** | **hijk** | qrst |

**Figure 3.3** : Example operation of word-size permute instruction.

### 3.1.4 wsfilter_a(C,K) (word-size filter computing instruction on AVX2)

wsfilter_a specialized word-size packed instruction calculates the filter values using shift value K on 32-byte chunks C. This instruction can be emulated in constant time by the following AVX2 intrinsics functions.

```
- K : Shift value according to the alphabet
$D \leftarrow$ _mm256_slli_epi64( $C,K$ )
$f \leftarrow$ _mm256_movemask_epi8( $D$ )
```

Shifting operation is performed by `_mm256_slli_epi64(a,i)` instruction which left shifts 256-bit data by input *i* while padding zeros and masking operation is performed by `_mm256_movemask_epi8(a)` instruction which creates 32-bit mask from msb of each 32 bytes stored as 256-bit register. The following diagram sketches the operation of filter computing using AVX2 intrinsics on 32-byte chunks $C^j$ of pattern and text.

Filter calculation has two main operations as shifting and masking of 32-byte blocks. Shifting operation is required to make the filter more distinguishable. If text characters are inside in the first 128 of ASCII table, msb of each character is 0 so all filters will become zero without shifting. The most informative bit of text characters should be determined to create distinguishing filters. A convenient method is taking into account only the alphabet (|Σ| bytes) with assuming the text characters have a uniform distribution. K shifting values of data sets which will be used for experimental evaluation of algorithms are given in the table 3.1.

**Table 3.1** : K shifting values of datasets.

| | Data Set | Σ | K Value |
|---|---|---|---|
| 1 | Genome Sequence | 4 | 5 |
| 2 | Protein Sequence | 20 | 7 |
| 3 | English Language Text | 128 | 7 |

Filter computing over 256-bit chunk (32 char) bit with representation is sketched in the diagram 3.4.

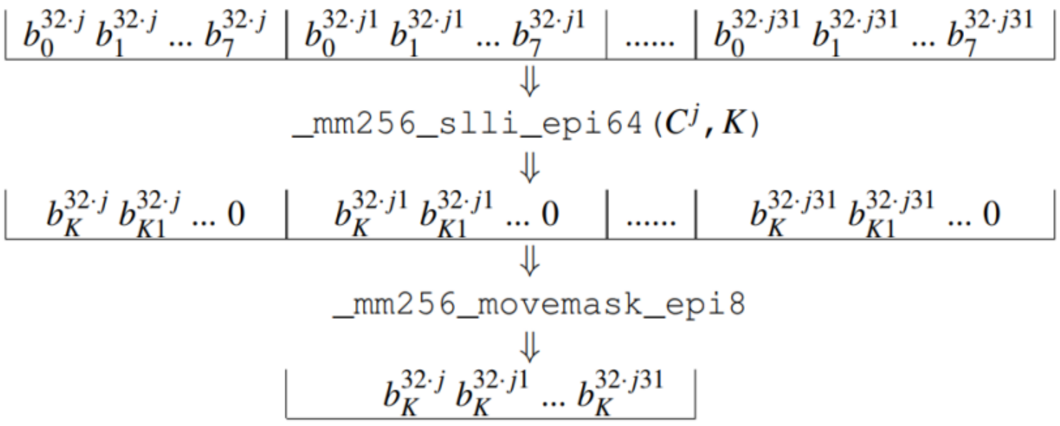

$b_0^{32\cdot j}\ b_1^{32\cdot j}\ \dots\ b_7^{32\cdot j}$ | $b_0^{32\cdot j1}\ b_1^{32\cdot j1}\ \dots\ b_7^{32\cdot j1}$ | ...... | $b_0^{32\cdot j31}\ b_1^{32\cdot j31}\ \dots\ b_7^{32\cdot j31}$
_mm256_slli_epi64($C^j$, $K$)
$b_K^{32\cdot j}\ b_{K1}^{32\cdot j}\ \dots\ 0$ | $b_K^{32\cdot j1}\ b_{K1}^{32\cdot j1}\ \dots\ 0$ | ...... | $b_K^{32\cdot j31}\ b_{K1}^{32\cdot j31}\ \dots\ 0$
_mm256_movemask_epi8
$b_K^{32\cdot j}\ b_K^{32\cdot j1}\ \dots\ b_K^{32\cdot j31}$

**Figure 3.4** : The sketch of filter computing.

An example operation of wsfilter_a instruction is given in the figure 3.5 where shifting value K is 7 and sample text is composed of 32 characters given as *"abcdefghijklmnopqrstuvwxyz 01234"* .

| Byte | 1 | 2 | 3 | 4 | 5 | 6 | ... | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ASCII | a | b | c | d | e | f | ... | ' ' | 0 | 1 | 2 | 3 | 4 |
| Dec | 97 | 98 | 99 | 100 | 101 | 102 | ... | 32 | 48 | 49 | 50 | 51 | 52 |

$\Downarrow$ `_mm256_slli_epi64(`$text$`,`$K$`)` $\Downarrow$

| D | 128 | 48 | 177 | 49 | 178 | 50 | ... | 128 | 16 | 152 | 24 | 153 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$\Downarrow$ `_mm256_movemask_epi8(`$D$`)` $\Downarrow$

| Filter | 1431654741 |
|---|---|

**Figure 3.5** : Example operation of the word size filter instruction

### 3.1.5 popcnt($a$) instruction

popcnt instruction corresponds to the `_mm_popcnt_u32(a)` instruction which counts the number of bits set to 1 in input $a$ which is unsigned 32-bit integer and returns the count value.

## 3.2 EPSMA Algorithms

EPSM(Exact Packed String Matching) algorithm is implemented using SSE intrinsics operated on 128-bit registers called as XMM [8]. In the EPSMA(EPSM on AVX2) is the new version of EPSM algorithm, AVX2 intrinsics are utilized which operate on 256-bit registers called as YMM. For discrete ranges of pattern lengths, three types of EPSMA algorithm are developed for most optimum solution in its range such as EPSMA-1, EPSMA-2 and EPSMA-3. These algorithms are essentially composed of word-size instructions aforementioned in section 3.1, descriptions and pseudo codes of developed algorithms are given below.

### 3.2.1 EPSMA-1

The EPSMA-1 algorithm is developed for short patterns therefore it is used for pattern lengths which are smaller than 8. The algorithm is based on wscmp_a instruction described above. It has two main phase; the preprocessing of the algorithm (lines 2-5) and the searching phase (lines 6-13) as shown in EPSMA-1 pseudo code, algorithm 1. It can be said that by taking into account description of wscmp_a, $p[0...m-1]$ has occurrence starting at position $j$ of $T_i$ if and only if $r_j = 1$. Occurrence count is calculated by popcnt instruction (line 10) and carry bits obtained by masking are stored (line 11) for comparison on the next loop. Besides, if there is a remaining part at the end of the text, the naive check method is applied for this text part (line 12-13).

### 3.2.2 EPSMA-2

The EPSMA-2 algorithm is designed by taking advantage of wsmatch_a instruction which implements the multiple SADs operation. It is convenient for pattern lengths from 8 to 32, while it could be used for greater pattern length, the performance of algorithm decreases experimentally. This algorithm benefits from the filtering technique, 2-stage filtering is applied sequentially using 8 characters (4+4) of pattern assigned in the preprocessing phase (lines 2-3).

**Algorithm 1** EPSMA-1 Algorithm Pseudo Code

```
 1: procedure EPSMA1( $p,m,t,n$ )
 2:     $n' \leftarrow 32 * (n/32)$ // last index divisible by 32
 3:     for $i \leftarrow 0$ to ( $m$ -1 ) do
 4:         for $j \leftarrow 0$ to (32-1) do
 5:             $B_i[j] \leftarrow p[i]$
 6:     for $i \leftarrow 0$ to ( $n'/32$ )- 1 do
 7:         for $j \leftarrow 0$ to ( $m$ - 1 ) do
 8:             $s_j \leftarrow$ wscmp_a( $T_i, B_j$ )
 9:             $r \leftarrow (s_j \ll j) \mid (carry_j \gg (32-j))$
10:             $count \leftarrow count +$ popcnt( $r$ )
11:             $carry_j \leftarrow s_j \,\&\, mask_j$
12:     for $j \leftarrow n' - 32$ to $n$ do
13:         check position at( $j - m$ ) // for last remaining part
```

At the first stage wsmatch_a instruction is applied to first 4 characters $p'_1$ (line 6). If there is a matching for this 4 characters $r$ value will be greater than zero, then first stage will be passed (line 7) and program flow will step into second stage. wsmatch_a instruction is applied again to second 4 characters $p'_2$ (line 9) after shuffling the text $T_i$ to make the data in the right order for new comparison. (line 8).

$r$ value which is greater than zero after wsmatch_a operation in second stage implies that there is a one at least or more matching for 8 characters. If pattern length ($m$) equals 8, the algorithm reports pattern occurrence easily (lines 11-12), otherwise the naive check is performed to possible positions starting at $i$*32 regarding $r$ (line 13).

In order to arrange the text data required for applying wsmatch_a instruction to second 16-byte chunk of text, wspermute_a instruction described in 3.1.3 is used (lines 14). 2-stage filtering operations are performed for the second part of each loop (lines 15-22) similarly applied as the previous part (lines 6-13). Additionally, when pattern length $m$ is between 16 and 32, 4 characters of the pattern are skipped while assigning characters in the preprocessing phase (line 2-3) in order to filter more selectively. So, second character assignment (line 3) is like that;

- $p'_2 \leftarrow p[8..11]$ for $16 \leq m$ 32

Finally, if the last chunk has a remaining part of text, naive check method will be applied for this part to attain complete matching.

**Algorithm 2** EPSMA-2 Algorithm Pseudo Code

```
1: procedure EPSMA2( $p,m,t,n$ )
2:     $p'_1 \leftarrow p[0..3]$
3:     $p'_2 \leftarrow p[4..7]$
4:     $idx \leftarrow$ _mm256_setr_epi32(1,2,3,0,3,4,5,0)
5:     for $i \leftarrow 0$ to ( $n/32$ )- 1  do
6:         $r \leftarrow$ wsmatch_a( $T_i$, $p'_1$ )
7:         if $r > 0$ then
8:             $S \leftarrow$ vpermd($T_i$, idx)
9:             $r \leftarrow$ wsmatch_a($S$, $p'_2$ )
10:            if $r > 0$ then
11:                if $m = 8$ then
12:                    report occurences at ($i$*32 + $r$)
13:                else check position at ($i$*32 + $r$)
14:        $S \leftarrow$ wspermute_a( $T_i$, $T_{i1}$ )
15:        $r \leftarrow$ wsmatch_a( $S$, $p'_1$ )
16:        if $r > 0$ then
17:            $S \leftarrow$ vpermd($S$, idx)
18:            $r \leftarrow$ wsmatch_a( $S$, $p'_2$ )
19:            if $r > 0$ then
20:                if $m = 8$ then
21:                    report occurences at ($i$*32+16 + $r$)
22:                else check position at ($i$*32+16 + $r$)
```

### 3.2.3 EPSMA-3

The EPSMA-3 algorithm uses the filtering approach inspiring by SSEF algorithm [9]. 16-bit filters are operated for filtering stage as in SSEF algorithm but in EPSMA-3 the calculation of filters are made on 256-bit data chunks, unlike SSEF. This algorithm utilizes wsfilter_a instruction which composed of shifting and masking operations as described in 3.1.4. After filter computing performed on 256-bit data separate filters $f_1$ and $f_2$, which have 14-bit filter length giving the best performance, are extracted from 32-bit filter $f$ (line 9 and 15). In the preprocessing phase (line 2-10) all possible filters of the pattern are computed and then stored as an array of filters in $FilterArray_1$ and $FilterArray_2$.

In the searching phase (lines 11-20), if filters $f_1$ and $f_2$ computed on text chunks (lines 14-15) exist in $FilterArray_1$ and $FilterArray_2$ respectively, the naive comparison will be applied separately (line 17 and line 19) to check whether there is an exact occurrence of the pattern. For shifting operation of filter computing, K values represented in table 3.1 are used in EPSMA-3. Furthermore, the loop of searching phase is unrolled by a factor of 2 in order to achieve optimal performance. EPSMA-3 algorithm is used for pattern lengths which are between 32 and 64 ( $32 \leq m \; 64$ ).

**Algorithm 3** EPSMA-3 Algorithm Pseudo Code

```
1:  procedure EPSMA3 ( $p,m,t,n$ )
2:      $L \leftarrow$ [ $m$/16 ] - 1
3:      $FilterArray_{1,2} \leftarrow \emptyset$
4:      $mask \leftarrow$ 0x3FFF
5:      $K \leftarrow$ a, $0 \leq a < 8$, according to the alphabet;
6:      for $i \leftarrow 0$ to $(16 \cdot L + 1)$ do
7:          $d \leftarrow$ _mm256_set_epi8 $(p_{i+31}, ..., p_i)$
8:          $f \leftarrow$ wsfilter_a( $d,K$ )
9:          $ftemp \leftarrow$ f $\gg 2$ ; $f_1 \leftarrow ftemp\&mask$ ; $f_2 \leftarrow ftemp \gg 16$
10:         $FilterArray_{1,2}[\, f_{1,2}] \leftarrow FilterArray_{1,2}[\, f_{1,2}] \cup i_{1,2}$
11:     while $i < N$ do
12:         if $L$ =2 then
13:             $T_i \leftarrow$ vperm2f128 _a( $T_i$, $T_{i+1}$, 32 )
14:         $f \leftarrow$ wsfilter_a($T_i$,$K$)
15:         $ftemp \leftarrow$ f $\gg 2$ ; $f_1 \leftarrow ftemp\&mask$ ; $f_2 \leftarrow ftemp \gg 16$
16:         for all $j \in FilterArray_1$ [ $f_1$] do
17:             check occurrence at( $t_{32 \cdot (i) - j}$ )
18:         for all $j \in FilterArray_2$ [ $f_2$] do
19:             check occurrence at( $t_{32 \cdot (iL) - j}$ )
20:         $i \leftarrow i + L$
```

## 3.3 SSEFA Algorithm

SSEFA (SSEF on AVX2) is a new variation of SSEF algorithm which has filter based approach composed of filtering and the verification phases. SSEFA is designed for exact matching of long patterns such as greater sizes than 64 ($64 \leq$m). All filter values of possible pattern alignments should be calculated to catch all location of matching candidates. For this purpose primarily, appropriate alignments on the given pattern are examined in the following part.

The zero-based address of the last 32-byte chunk of pattern not including zero padding is represented by $L$ symbol as $L = \lfloor m/32 \rfloor - 1$. For instance, let's assume m=120, 32-byte chunks of the pattern are like that $R = R^0R^1R^2R^3$ in this case. 24 bytes of last chunk $R^3$ are composed of the pattern, therefore remaining 8 bytes are padded with zero and $L$ value becomes $L = \lfloor 120/32 \rfloor$-1=2. If $h$ is defined as $0 \leq h < \lfloor$ N/L $\rfloor$, the chunks which filter computing is performed can be represented such as $Y^{h\cdot L+L}$. If there is a proper alignment of the pattern from $Y^{h\cdot L}$ and $Y^{h\cdot L+(L-1)}$ according to the filter value, naive comparison on remaining part of the candidate text will be applied to find possible matching. Appropriate alignments of pattern bytes are sketched in the following diagram 3.6 when computing the filters on 32-byte chunks from $Y^i$ where $i = h \cdot L$ and all bytes of last chunk represented by $Y^{i+L}$ are filled with patterns.

| Chunks | $Y^i$ | $Y^{i+1}$ | | ... | $Y^{i+L\text{-}1}$ | | $Y^{i+L}$ | | |
|---|---|---|---|---|---|---|---|---|---|
| Bytes | $t_{32\cdot i}$ ... | $t_{32\cdot(i+1)}$ | ... | ... | $t_{32\cdot(i+L\text{-}1)}$ | ... | $t_{32\cdot(i+L)}$ | ... | $t_{32\cdot(i+L)+31}$ |
| Pattern | | | | | | | | | |
| $t_{32\cdot i}$ | $p_0$ ... | $p_{32}$ | ... | ... | $p_{32\cdot(L\text{-}1)}$ | ... | $p_{32\cdot L}$ | ... | $p_{32\cdot L+31}$ |
| $t_{32\cdot i+1}$ | $p_0$ ... | $p_{31}$ | ... | ... | $p_{32\cdot(L\text{-}1)\text{-}1}$ | ... | $p_{32\cdot L\text{-}1}$ | ... | $p_{32\cdot L+30}$ |
| | .................................... | | | | | | | | |
| $t_{32\cdot i+31}$ | ... $p_0$ | $p_1$ | ... | ... | $p_{32\cdot(L\text{-}1)\text{-}31}$ | ... | $p_{32\cdot L\text{-}31}$ | ... | $p_{32\cdot L}$ |
| | .................................... | | | | | | | | |
| $t_{32\cdot(i+L)\text{-}2}$ | | | | | $p_0$ | $p_1$ | $p_2$ | ... | $p_{33}$ |
| $t_{32\cdot(i+L)\text{-}1}$ | | | | | | $p_0$ | $p_1$ | ... | $p_{32}$ |

**Figure 3.6** : Appropriate pattern alignment.

Fundamentally, SSEFA algorithm has two phases as preprocessing (lines 2-10) and searching (lines 11-19) as depicted in pseudo-code 4. The preprocessing phase includes the initializations of variables and the calculation of filter values over the given pattern. wsfilter_a instruction explained in 3.1.4 is used (line 8) to calculate 32-bit filter $f$ values on 256-bit data chunks which are created by `_mm256_set_epi8` (line 7) intrinsic with pattern bytes. In the searching phase, the outer loop operates on 32-byte chunks $Y^i$ of text $T$ in steps of $L$ where $i{=}h \cdot L{+}L$ and $0 \leq h < \lfloor \text{N/L} \rfloor$. Also distinguishing bit position is assigned to $K$ variable used for shifting while applying the wsfilter_a instruction.

After wsfilter_a operation (line 8 and 12), filter $f$ is composed of 32-bit data, therefore, the filter can get a decimal value between 0 and $2^{32}$ (4294967295:4GB). Filter values are used as indexes while creating the filter vectors (line 10) and the size of the filter vector is determined by the maximum value of the filter $f$ can get. If 32-bit filters are used, vector size becomes extremely large as 4GB, in that case, memory access latencies in searching phase will cause the performance bottleneck. Filters vector could not fit in lower level caches and memory access time becomes a constraint in overall matching performance. Memory access analysis is applied using hardware event metrics on Intel Vtune Amplifier to detect memory issues and find optimal filter length. Detailed descriptions of memory analysis and optimization are given in the next section named as "Cache Profiling and Optimization" 3.4.

**Algorithm 4** SSEFA Algorithm Pseudo Code

```
1:  procedure SSEFA ( $p,m,t,n$ )
2:      $L \leftarrow$ [ $m/32$ ] - 1
3:      $FilterArray_{1,2} \leftarrow \emptyset$
4:      $shift \leftarrow 18$ ; $mask \leftarrow$ 0x3FFF
5:      $K \leftarrow$ a, $0 \le a < 8$, according to the alphabet;
6:      for $i \leftarrow 0$ to $(32 \cdot L + 1)$ do
7:          $d \leftarrow$ _mm256_set_epi8 $(p_i+31, .., p_i)$
8:          $f \leftarrow$ wsfilter_a($d,K$)
9:          $f_1 \leftarrow$ f $\gg shift$ ; $f_2 \leftarrow$ f & $mask$
10:         $FilterArray_{1,2}[f_{1,2}] \leftarrow FilterArray_{1,2}[f_{1,2}] \cup i_{1,2}$
11:     while $i < N$ do
12:         $f \leftarrow$ wsfilter_a($T_i,K$)
13:         $f_1 \leftarrow$ f $\gg shift$
14:         for all $j \in FilterArray_1 [f_1]$ do
15:             $f_2 \leftarrow$ f & $mask$
16:             for all $j \in FilterArray_2 [f_2]$ do
17:                 if $P$ $[t_{32\cdot(i-L)+j} \ldots t_{32\cdot(i-L)+j+m-1}]$ then
18:                     pattern occurrence at( $t_{32\cdot(i-L)+j}$ )
19:         $i \leftarrow i + L$
```

As a result of memory access profiling, 2-stage filtering approach with 14-bit filter length gives the best performance. 14-bit wide filters $f_1$ and $f_2$ are extracted from 32-bit filter $f$ (line 9,12 and 15). Filter $f_1$ is obtained by shifting operation with $shift$ value (18) so $f_1$ gets 14 high-order bits (left-most bits) of 32-bit $f$. On the other hand, filter $f_2$ is obtained by masking operation using *mask* (0x3FFF) and $f_2$ gets 14 low-order bits (right-most bits) of 32-bit $f$. Reduced 14-bit filters $f_1$ and $f_2$ exists as indexes in the arrays named as $FilterArray_1$ and $FilterArray_2$ respectively (line 10), then these filter arrays will be used for searching the filters calculated on text chunks (line 14 and 16). $FilterArray_1$ is utilized like a guard for first level filtration so $FilterArray_1[i]$ value is set 1 where filter exists otherwise set to 0. $FilterArray_2$ is used as the secondary filtering to decrease the number of verification of exact pattern and the possible beginning position of the pattern in the text is assigned to $FilterArray_2[i]$ regarding filter, otherwise set to 0. If 2-stage filtering is passed successfully over chunk $Y^i$, a full verification will be performed between P and $t_{32\cdot(i\text{-}L)+j}...t_{32\cdot(i\text{-}L)+j+m\text{-}1}$ using j value of $FilterArray_2[f]$ where $0 \leq j \;\; 32 \cdot L$ .

## 3.4 Profiling and Optimization

Performance analysis of HPC(high-performance computing) system is a crucial point while developing efficient applications for modern architectures. Analysis results can guide software developers for tuning the algorithms and improving the algorithm performance. In this section, firstly Intel Vtune Performance Analyzer is explained and then hardware-based performance measurements of the developed algorithm and applied optimizations on SSEFA are presented.

### 3.4.1 Intel Vtune Amplifier

Intel Vtune Amplifier is integrated performance analyzer tool used to detect hardware bottlenecks for HPC application or system on Intel modern microarchitectures. Vtune Amplifier uses hardware data collectors to show the performance issues in a user-friendly format, so it provides focusing on code tuning effort and achieving the best performance improvement in the least amount of time. Vtune can be run with GUI(Graphical User Interface) or CLI(command-line interface) on Linux and Windows platform, also performance analyses can be made by the remote terminal over the network.

**Table 3.2** : Properties of Intel Vtune Amplifier Collectors.

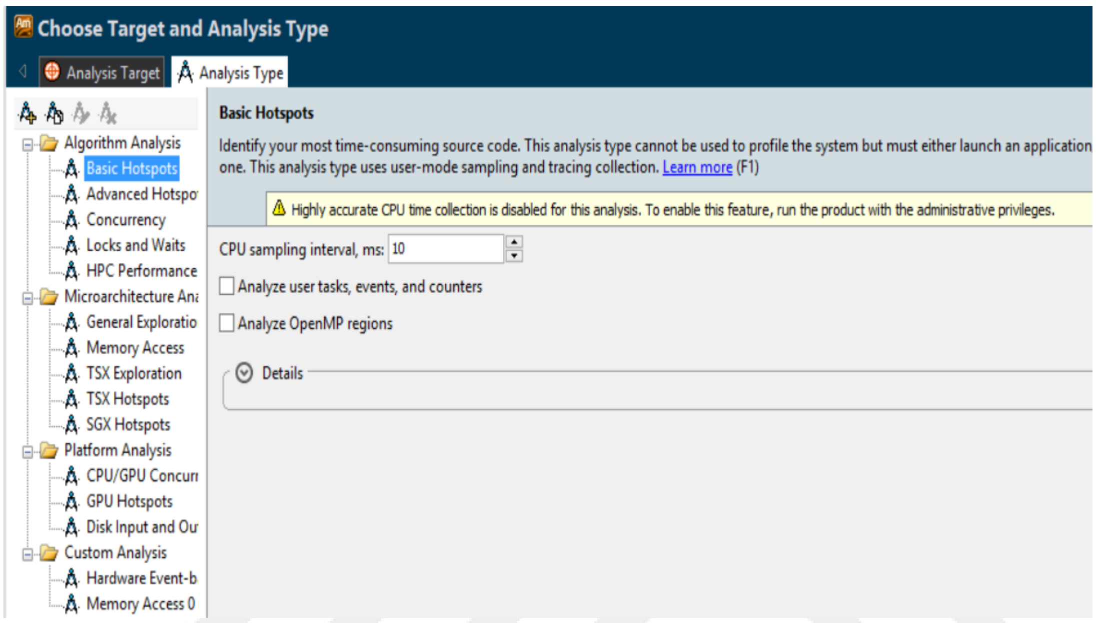
| Hardware Collector | Software Collector |
|---|---|
| Uses the on chip Performance Monitoring Unit (PMU) | Uses OS interrupts |
| Optionally collects call stacks | Call stacks show calling sequence |
| Requires a driver | No driver required |
| 1ms sampling interval, low overhead | 10ms sampling interval |
| Advanced Hotspots | Basic Hotspots |
| Microarchitecture and Platform Analysis | Threading: Concurrency, Lock/Waits etc. |

Analyzes of VTune Amplifier are based on sampling performance data collected with Hardware Collector or Software Collector, these collection types also named as Hardware Event-based Sampling Collection and User-Mode Sampling/Tracing Collection [18]. Basic properties and analysis types are presented in the table 3.2, later detailed descriptions of analysis types are given.

- *Hotspots Analysis*: This analysis is used as a starting point to analyze your algorithm. It shows application flow and functions that took the most CPU time to execute. It is a practical way to identify performance-critical code sections in the application and explore memory consumption (RAM) over time with memory objects.
- *Parallelism Analyzes*: It can be used for parallel compute-sensitive applications for overall performance analysis. Threading analysis provide Effective CPU Utilization metrics for measurement of threading efficiency such as Total Thread Count, Wait Time with Poor CPU Utilization, Spin and Overhead Time.
- *Microarchitecture Analyzes*: It helps to detect the issues affecting the performance related to hardware-level. "Microarchitecture Exploration" analysis type is a starting point of hardware-level analysis. "Memory Access" analysis type gives a set of metrics that show issues about memory access such as Memory Bound, Loads, Stores, LLC (Last-level cache) Miss Count, Average Latency.
- *The Platform Analyzes*: These analysis types are used for monitoring CPU, GPU system and power usage for the application. Platform analysis group includes various subtypes such as CPU/GPU Concurrency, System Overview, Input and Output analysis, CPU/FPGA Interaction, Platform Profiler etc.
- *Source Code Analysis*: Performance problem associated with the source code and exact machine instruction(s) can be identified using this type of analysis. Source codes and related assembly instructions and CPU times are presented in the same pane to quickly identify the hotspot lines.
- *Custom Analysis*: New custom analysis can be created using the data collectors provided by the VTune Amplifier or any other custom collector. PMU events

monitored by the Vtune Amplifier can be added to custom analysis and some options are configurable if required like "CPU sampling interval value".

A view of analysis types on Vtune Amplifier GUI is depicted in the figure 3.7, analysis options are grouped on the left of GUI for selection.


Choose Target and Analysis Type

Analysis Target | Analysis Type

- Algorithm Analysis
  - Basic Hotspots
  - Advanced Hotspo
  - Concurrency
  - Locks and Waits
  - HPC Performance
- Microarchitecture Ana
  - General Exploratio
  - Memory Access
  - TSX Exploration
  - TSX Hotspots
  - SGX Hotspots
- Platform Analysis
  - CPU/GPU Concurr
  - GPU Hotspots
  - Disk Input and Ou
- Custom Analysis
  - Hardware Event-b
  - Memory Access 0

**Basic Hotspots**

Identify your most time-consuming source code. This analysis type cannot be used to profile the system but must either launch an application one. This analysis type uses user-mode sampling and tracing collection. Learn more (F1)

⚠ Highly accurate CPU time collection is disabled for this analysis. To enable this feature, run the product with the administrative privileges.

CPU sampling interval, ms: 10

☐ Analyze user tasks, events, and counters

☐ Analyze OpenMP regions

Details

**Figure 3.7** : Analysis types of Vtune Amplifier.

### 3.4.2 Memory analysis and cache optimization

Memory Access Analysis identifies memory-related issues especially high memory access time when data can not fit in the L1 or L2 caches.

**Table 3.3** : Intel Microarchitecture Hardware Events (uops:micro-operations).

| Hardware Event Name | Definition |
|---|---|
| MEM_LOAD_UOPS_RETIRED.L1_HIT: | Retired load uops with L1 cache hits as data sources. |
| MEM_LOAD_UOPS_RETIRED.L1_MISS: | Retired load uops missed L1 cache as data sources. |
| MEM_LOAD_UOPS_RETIRED.L2_HIT: | Retired load uops with L2 cache hits as data sources. |
| MEM_LOAD_UOPS_RETIRED.L2_MISS: | Retired load uops missed L2. Unknown data source excluded. |
| MEM_LOAD_UOPS_RETIRED.L3_HIT: | Retired load uops with L3 cache hits as data sources. |
| MEM_LOAD_UOPS_RETIRED.L3_MISS: | Retired load uops missed L3. Excludes unknown data source. |
| INST_RETIRED.ANY: | counts the number of instructions retired from execution. |
| CPU_CLK_UNHALTED.THREAD: | counts the number of cycles while the thread is not in a halt. |

Hardware Event-Based Sampling (EBS) Analysis, also known as Performance Monitoring Counter (PMC) analysis, is selected for collecting event metrics values related to caches from the microarchitecture. Hardware events mentioned in table 3.3 particularly related to L1 and L2 caches are added to memory analysis on Vtune Amplifier Tool.

**Cache Miss Rate:** Cache miss is a condition where the data requested for processing by the application is not found in the related cache memory. Cache miss rate is a critical parameter on the measurement of cache performance and it can be expressed as equation 3.1 in terms of hardware event metrics collected on Vtune [19].

$$L_i MissRate \equiv \frac{MEM\_LOAD\_UOPS\_RETIRED.L_i\_MISS}{INST\_RETIRED.ANY} \qquad i{:}\ 1,2,3 \tag{3.1}$$

High cache miss rate implies that the advantage of the cache memory performance cannot be utilized exactly. The processor waste more time accessing the requested data if data doesn't exist in lower level caches.

**CPI Rate:** CPI (Cycles per Instruction), is a master performance metric for the analysis with hardware event-based sampling collections on Intel Vtune Amplifier. It indicates how many cycles have been executed to complete related instruction. Modern processor architectures can execute four instructions per cycle therefore theoretical best CPI value is 0.25. In general, high CPI value shows that performance decrease of the application which could be caused by issues such as memory stalls, long latency instructions, branch misprediction or instruction starvation. CPI value can be reduced as a result of optimizations using hardware-related metrics which identifies what is causing high CPI.

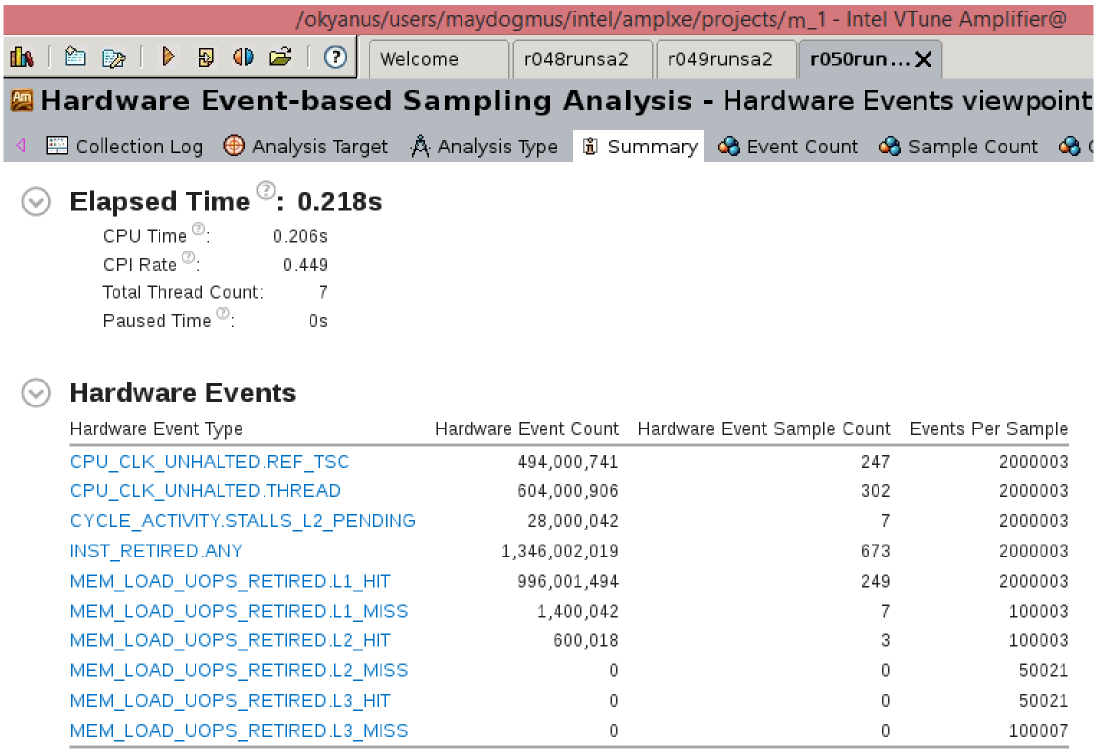
#### 3.4.2.1 Filter length analysis

In order to find the optimal filter vector length of SSEFA Algorithm, Hardware Event-Based Sampling (EBS) Analysis is applied while increasing the vector lengths from 4KB to 4GB. L3 cache hits and misses are removed from results because these values are zero for all conditions. in the figure 3.8, a sample results view of Hardware Event-Based Sampling (EBS) Analysis on Vtune Amplifier GUI is depicted. Elapsed Time, CPU Time, CPI Rate and Hardware Event Count are given in the Summary section of Vtune Amplifier GUI.

**Table 3.4** : Hardware Event values for various vector lengths.

| **List Size** | **4KB** | **16KB** | **64KB** | **256KB** | **1MB** | **4MB** |
|---|---|---|---|---|---|---|
| CPU_CLK_UNHALTED.THREAD | 612,000,918 | 610,00,915 | 612,000,918 | 612,000,918 | 622,000,933 | 652,000,978 |
| INST_RETIRED.ANY | 1,352,002,028 | 1,354,002,025 | 1,350,002,025 | 1,354,002,031 | 1,356,002,034 | 1,358,002,037 |
| MEM_LOAD_UOPS_RETIRED.L1_HIT | 1,004,001,506 | 992,001,488 | 1,000,001,500 | 1,000,001,500 | 1,004,001,506 | 996,001,494 |
| MEM_LOAD_UOPS_RETIRED.L1_MISS | 2,400,072 | 2,300,072 | 2,600,078 | 2,600,078 | 2,640,078 | 2,675,260 |
| MEM_LOAD_UOPS_RETIRED.L2_HIT | 1,200,036 | 1,600,048 | 1,000,030 | 800,024 | 600,018 | 600,018 |
| MEM_LOAD_UOPS_RETIRED.L2_MISS | 0 | 0 | 0 | 0 | 0 | 0 |

| **List Size** | **16MB** | **64MB** | **256MB** | **1GB** | **4GB** |
|---|---|---|---|---|---|
| CPU-CLK-UNHALTED.THREAD | 732,001,098 | 844,001,266 | 2,222,003,333 | 6,928,010,392 | 25,738,038,607 |
| INST-RETIRED.ANY | 1,368,002,052 | 1,426,002,139 | 1,640,002,460 | 2,500,003,750 | 5,942,008,913 |
| MEM-LOAD-UOPS-RETIRED.L1-HIT | 992,001,488 | 1,012,001,518 | 1,020,001,530 | 1,052,00,578 | 1,220,001,830 |
| MEM-LOAD-UOPS-RETIRED.L1-MISS | 2,800,078 | 3,200,096 | 4,400,132 | 8,600,258 | 25,400,762 |
| MEM-LOAD-UOPS-RETIRED.L2-HIT | 600,018 | 1,000,030 | 1,200,036 | 3,000,090 | 9,000,270 |
| MEM-LOAD-UOPS-RETIRED.L2-MISS | 0 | 0 | 0 | 0 | 0 |



Hardware Event-based Sampling Analysis - Hardware Events viewpoint

Elapsed Time: 0.218s

CPU Time: 0.206s
CPI Rate: 0.449
Total Thread Count: 7
Paused Time: 0s

Hardware Events

| Hardware Event Type | Hardware Event Count | Hardware Event Sample Count | Events Per Sample |
|---|---|---|---|
| CPU_CLK_UNHALTED.REF_TSC | 494,000,741 | 247 | 2000003 |
| CPU_CLK_UNHALTED.THREAD | 604,000,906 | 302 | 2000003 |
| CYCLE_ACTIVITY.STALLS_L2_PENDING | 28,000,042 | 7 | 2000003 |
| INST_RETIRED.ANY | 1,346,002,019 | 673 | 2000003 |
| MEM_LOAD_UOPS_RETIRED.L1_HIT | 996,001,494 | 249 | 2000003 |
| MEM_LOAD_UOPS_RETIRED.L1_MISS | 1,400,042 | 7 | 100003 |
| MEM_LOAD_UOPS_RETIRED.L2_HIT | 600,018 | 3 | 100003 |
| MEM_LOAD_UOPS_RETIRED.L2_MISS | 0 | 0 | 50021 |
| MEM_LOAD_UOPS_RETIRED.L3_HIT | 0 | 0 | 50021 |
| MEM_LOAD_UOPS_RETIRED.L3_MISS | 0 | 0 | 100007 |

**Figure 3.8** : Sample results of EBS analysis on Vtune Amplifier.

Hardware Event Metrics values measured over 100 test samples on SSEFA algorithm are shown in the table 3.4. L1 Miss Rate calculated with equation 3.1, CPI Rate and Elapsed Time values are given for different lengths in the table 3.5.

**Table 3.5** : Time and rate values for various vector lengths.

| **List Size** | **4KB** | **16KB** | **64KB** | **256KB** | **1MB** | **4MB** | **16MB** | **64MB** | **256MB** | **1GB** | **4GB** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Elapsed Time [ms] | 0.218 | **0.210** | 0.224 | 0.215 | 0.222 | 0.234 | 0.264 | 0.360 | 0.782 | 2.413 | 8.927 |
| CPI-Rate | 0.453 | **0.450** | 0.454 | 0.452 | 0.459 | 0.480 | 0.535 | 0.718 | 1.355 | 2.771 | 4.532 |
| L1 Miss Rate % | 1.775 | **1.698** | 1.925 | 1.920 | 1.946 | 1.969 | 2.046 | 2.244 | 2.683 | 3.440 | 4.274 |

It can be seen in the table 3.5 the best CPI rate is 0.450 and L1 miss rate is 1.698 at the size of 16384 where the mask is composed of 14-bit 1. However, if the only single 14-bit filter is used at filtration, there will be fewer filtering on match candidates of text chunks. For the purpose of increasing the selectivity of filtering operation, two-stage filtering method (applying 2x14-bit filters) is used therefore the number of full verification decreases.

Reduced filters using in 2-stage filtering are $FilterArray_1$ and $FilterArray_2$ as shown in pseudo-code of SSEFA and both of them are composed of distinct bits from separate halves of the main 32-bit filter. $FilterArray_1$ contains filters consisted of highest 14 bits (leftmost) of 32-bit filter whereas filters of $FilterArray_2$ consist of lowest 14 bits (rightmost). Formally, reduced filters ( two 14 bit filters f1 and f2) are represented in terms of bit items like that;

32-bit filter , $f_i : b^i_0 b^i_1 b^i_2 ... b^i_{31}$

reduced filter 1 , $f_{1i} : b^i_0 b^i_1 b^i_2 ... b^i_{13}$

reduced filter 2 , $f_{2i} : b^i_{18} b^i_{19} b^i_{20} ... b^i_{31}$

These reduced filters are represented in line 8-9 of pseudo-code of SSEFA.

### 3.4.2.2 Data type analysis

Filters are stored as a linked-list in SSEF algorithm [20] however accessing an element maybe a little slower due to using "nodes" on linked-list. Links are allocated at random separate locations which can cause cache misses while trying to access the pointer. On the other hand, if number of the filters is fixed, the array structure can be used to hold filters which can allow fast random access. Data of array structure are stored in contiguous memory therefore it improves cache spatial locality. In the table 3.6, hardware event metric values collected from Intel Vtune Amplifier are given in table 3.6 when filters are stored in the array and linked-list structures.

**Table 3.6** : Hardware event metrics for array and linked-list structures.

| Structure / Patlen | Array, 64 | Linked-List, 64 | Array, 512 | Linked-List, 512 |
|---|---|---|---|---|
| CPU-CLK-UNHALTED.THREAD | 598,000,897 | 612,000,918 | 592,000,909 | 606,000,909 |
| INST-RETIRED.ANY | 1,350,002,025 | 1,354,002,031 | 1,344,002,016 | 1,386,002,019 |
| MEM-LOAD-UOPS-RETIRED.L1-HIT | 992,001,488 | 996,001,500 | 996,001,488 | 996,001,494 |
| MEM-LOAD-UOPS-RETIRED.L1-MISS | 2,600,078 | 2,800,072 | 1,400,048 | 1,600,048 |
| MEM-LOAD-UOPS-RETIRED.L2-HIT | 1,800,054 | 1,600,048 | 600,018 | 800,024 |
| MEM-LOAD-UOPS-RETIRED.L2-MISS | 0 | 0 | 0 | 0 |

CPI Rate, Elapsed Time values and calculated L1 Miss Rate are presented for the array and linked-list structures in the table 3.7.

**Table 3.7** : Time and rate values for array and linked-list structures.

| Structure / Patlen | Array, 64 | Linked-List, 64 | Array, 512 | Linked-List, 512 |
|---|---|---|---|---|
| Elapsed Time | **0.205** | 0.222 | **0.191** | 0.202 |
| CPI-Rate | **0.443** | 0.455 | **0.440** | 0.453 |
| L1Miss Rate | **1.925** | 2.067 | **1.041** | 1.154 |

As can be seen from the table 3.7, using array structure gives the better result than linked-list type when memory access is the major factor in processing such as accessing a value over fixed numbers of filters. Also, the linear probing method is applied to the array structure for collision handling of filter values. Linear probing method defines as: if related spot is occupied, continue moving through the array structure until a free spot will be found. This method also known as open-addressing hashing strategy.

# 4. EXPERIMENTAL RESULTS

Algorithms have been implemented in the C programming language and SMART (String Matching Algorithms Research Tool) [21] is used to compare the performances of the algorithms.

**Table 4.1** : Test platform.

| Component | Property |
|---|---|
| CPU | Intel Xeon E5-2680 v4 |
| L1d cache | 32KB, 64B line size, 4 Latency Cycles |
| L1i cache | 32KB, 64B line size, 4 Latency Cycles |
| L2 cache | 256KB, 64B line size, 12 Latency Cycles |
| L3 cache | 35MB, 64B line size, 40+ Latency Cycles |
| OS | CentOS 7 x86-64 |
| GCC Version | 4.8.5 20150623 (UHeM System Default Version) |

GCC compiler is used with std=gnu99 mode and full optimization option is selected by -O3 flag. Three dataset types are used; genome sequence ( $|\Sigma|$=4), protein sequence ( $|\Sigma|$=20) and natural language text (English language, $|\Sigma|$=128 ) provided by the Smart research tool and dataset sizes are 200MB. Test data is loaded to the memory in the context of 32-byte aligned by union structure using __m256i AVX2 data type. All tests run over 100 times by setting pset (size of the set of patterns) is 100. Algorithms using in comparison are selected by scanning all existing algorithms on SMART Tool. Only the best result of algorithms is presented using q-grams and these q values are reported as apices. Search times of algorithms at result tables are expressed in milliseconds and best results of each pattern length have been boldfaced.

## 4.1 Results for EPSMA Algorithms (patlen<64)

The performance of EPSMA algorithm is compared with the following algorithms on SMART platform.

- BNDM: Backward Nondeterministic DAWG Matching by Navarro and Raffinot [22];
- BNDMq: The Backward DAWG Matching algorithm with q-grams [4];

- BSDMq: Backward-SNR-DAWG-Matching (BSDM) by Faro and Lecroq [23];
- EBOM: the Extended Backward Oracle Matching algorithm [24];
- EPSM: fast packed string matching for short patterns by Faro and Külekci [8];
- FSBNDMq: Forward Simplified BNDM using q-grams by Peltola and Tarhio [25];
- MEMCMP: direct usage of "memcmp" function of C library;
- N32-freq: compares 32 characters using AVX2 where comparison order given by nondecreasing probability of pattern symbols [13];
- N32-fixed: compares 32 characters using AVX2, comparison order is fixed [13];
- SKIPq: combination of Skip-Search and the Hashq algorithms [26];
- SBNDMq: implementation of the Simplified BNDM with q-grams (SBNDMq) [4];
- TVSBS: combination of Berry Ravindran and SSABS algorithms [27];
- UFNDMq: implementation of the Shift Or with q-grams algorithm [4] ;
- WFRq: efficient algorithm based on a weak factor recognition and hashing [28];

**Table 4.2** : Running times for genome sequence when pattern length < 64.

| **m** | **2** | **4** | **6** | **8** | **10** | **12** | **16** | **20** | **24** | **28** | **32** | **40** | **48** | **56** | **62** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BNDMq | 194.99$^{(2)}$ | 158.12$^{(2)}$ | 87.28$^{(4)}$ | 56 63$^{(4)}$ | 49.15$^{(4)}$ | 47.23$^{(4)}$ | 37.81$^{(4)}$ | 36.08$^{(4)}$ | 32.94$^{(6)}$ | 32.38$^{(6)}$ | 31.40$^{(6)}$ | 31.05$^{(6)}$ | 30.65$^{(6)}$ | 32.41$^{(6)}$ | 30.54$^{(6)}$ |
| BOM2 | | | 227.24 | 164.99 | 148.31 | 147.18 | 109.22 | 103.47 | 85.16 | 75.58 | 69.45 | 61.32 | 54.52 | 56.05 | 48.16 |
| BSDMq | 182 02$^{(2)}$ | 96 07$^{(3)}$ | 72 20$^{(4)}$ | 49 74$^{(4)}$ | 44 61$^{(4)}$ | 43 72$^{(4)}$ | 35 57$^{(6)}$ | 33 64$^{(6)}$ | 31 65$^{(7)}$ | 31 45$^{(6)}$ | 31 09$^{(7)}$ | 30 46$^{(7)}$ | 29 44$^{(7)}$ | 30 94$^{(7)}$ | 29 79$^{(7)}$ |
| EBOM | 171.94 | 132.41 | 142.37 | 116.69 | 114.76 | 114.88 | 88.72 | 84.24 | 70.44 | 64.97 | 60.60 | 54.54 | 48.99 | 50.48 | 43.94 |
| EPSM | 31.07 | 33.29 | 43.81 | 42.81 | 44.22 | 47.26 | 33.04 | 34.03 | 31.94 | 30.91 | 29.40 | 30.00 | 29.68 | 30.65 | 29.08 |
| EPSMA | **26.86** | **28.23** | **31.25** | **29.22** | **30.94** | **31.96** | 32.86 | **31.78** | **29.88** | **28.27** | **27.84** | **28.34** | **27.31** | **27.63** | **27.49** |
| FSBNDM | | 212.29 | 166. 15 | 111.66$^{(4)}$ | 94.04$^{(2)}$ | 90.39$^{(W2)}$ | 65.43$^{(W2)}$ | 60 97$^{(W2)}$ | 50.98$^{(W2)}$ | 46.68$^{(W2)}$ | 44 15$^{(W2)}$ | 44.26$^{(W2)}$ | 43.44$^{(W2)}$ | 48 38$^{(W2)}$ | 43.99$^{(W2)}$ |
| FSBNDMq | 204 41$^{(20)}$ | 106.79$^{(31)}$ | 74 91$^{(41)}$ | 52.38$^{(41)}$ | 47.37$^{(41)}$ | 46.07$^{(41)}$ | 37.48$^{(62)}$ | 35.27$^{(62)}$ | 31.88$^{(61)}$ | 31.63$^{(62)}$ | 30.99$^{(61)}$ | 31.00$^{(61)}$ | 30.22$^{(61)}$ | 32.52$^{(61)}$ | 30.54$^{(61)}$ |
| MEMCMP | 540.10 | 640.99 | 740.05 | 810.21 | 820.17 | 720.05 | 680.59 | 760.75 | 860.52 | 890.49 | 820.09 | 890.16 | 760.37 | 790.00 | 772.41 |
| N32-freq | 29 17$^{(2)}$ | 32 10$^{(5)}$ | 31 93$^{(5)}$ | 30 84$^{(5)}$ | 32 22$^{(5)}$ | 33 02$^{(5)}$ | 30 95$^{(5)}$ | 32 50$^{(7)}$ | 30 91$^{(5)}$ | 31 72$^{(5)}$ | 31 65$^{(5)}$ | 31 58$^{(5)}$ | 30 75$^{(5)}$ | 33 01$^{(5)}$ | 31 19$^{(5)}$ |
| N32-fixed | 29 06$^{(2)}$ | 32 05$^{(5)}$ | 31 91$^{(5)}$ | 30 50$^{(5)}$ | 31 84$^{(5)}$ | 32 65$^{(5)}$ | **30.41** $^{(5)}$ | 32 89$^{(5)}$ | 30 06$^{(5)}$ | 31 24$^{(5)}$ | 31 13$^{(5)}$ | 31 36$^{(5)}$ | 30 43$^{(5)}$ | 32 65$^{(5)}$ | 30 86$^{(5)}$ |
| SBNDMq | 183.29$^{(2)}$ | 139.13$^{(2)}$ | 86.20$^{(4)}$ | 56 61$^{(4)}$ | 49.61$^{(4)}$ | 47.53$^{(4)}$ | 38.57$^{(5)}$ | 37.17$^{(4)}$ | 34.08$^{(4)}$ | 32.84$^{(6)}$ | 31.83$^{(6)}$ | 31.28$^{(6)}$ | 30.32$^{(6)}$ | 32.53$^{(6)}$ | 30.69$^{(6)}$ |
| SKIPq | 173.55$^{(2)}$ | 116.25$^{(3)}$ | 78.29$^{(4)}$ | 52 72$^{(4)}$ | 46.86$^{(4)}$ | 45.31$^{(4)}$ | 36.81$^{(6)}$ | 34.87$^{(6)}$ | 31.55$^{(6)}$ | 31.65$^{(6)}$ | 31.60$^{(6)}$ | 31.45$^{(7)}$ | 29.32$^{(6)}$ | 30.92$^{(7)}$ | 29.30$^{(7)}$ |
| SSEF | – | – | – | – | – | – | – | – | – | – | 58.74 | 49.68 | 35.92 | 37.92 | 36.40 |
| TVSBS-W8 | – | – | | 196.23 | 175.56 | 177.76 | 131.76 | 128.31 | 99.42 | 101.02 | 96.73 | 87.72 | 84.12 | 91.63 | 80.43 |
| UFNDMq | – | – | | 61 04$^{(8)}$ | 53.96$^{(8)}$ | 52. 16$^{(8)}$ | 40.41$^{(8)}$ | 40.00$^{(8)}$ | 36.21$^{(8)}$ | 35.05$^{(8)}$ | 35.00$^{(8)}$ | 35.01$^{(8)}$ | 34.06$^{(8)}$ | 37.10$^{(8)}$ | 34.46$^{(8)}$ |
| WFRq | 213.50$^{(2)}$ | 153.74$^{(2)}$ | 85.73$^{(3)}$ | 58 11$^{(4)}$ | 51.76$^{(5)}$ | 46.58$^{(4)}$ | 37.42$^{(5)}$ | 34.91$^{(4)}$ | 33.43$^{(5)}$ | 31.99$^{(5)}$ | 30.97$^{(5)}$ | 30.22$^{(5)}$ | 29.02$^{(5)}$ | 30.41$^{(5)}$ | 28.97$^{(5)}$ |

**Table 4.3** : Running times for protein sequence when pattern length < 64.

| **m** | **2** | **4** | **6** | **8** | **10** | **12** | **16** | **20** | **24** | **28** | **32** | **40** | **48** | **56** | **62** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BNDMq | 121 17$^{(2)}$ | 63.19$^{(2)}$ | 45.54$^{(2)}$ | 37.48$^{(2)}$ | 37.72$^{(2)}$ | 33.01$^{(2)}$ | 34.75$^{(2)}$ | 32.38$^{(4)}$ | 31.83$^{(4)}$ | 31.56$^{(4)}$ | 30. 15$^{(4)}$ | 29.37$^{(2)}$ | 29.23$^{(2)}$ | 29.64$^{(2)}$ | 29.12$^{(2)}$ |
| BOM2 | 156.24 | 134.95 | 98.06 | 91.83 | 94.67 | 76.04 | 69.77 | 52.27 | 45.78 | 45.73 | 38.28 | 33.73 | 31.68 | 31.26 | 29.50 |
| BSDMq | 99.62$^{(2)}$ | 59.44$^{(2)}$ | 48.02$^{(3)}$ | 38.42$^{(3)}$ | 37.79$^{(3)}$ | 33.21$^{(3)}$ | 33.97$^{(4)}$ | 32.78$^{(4)}$ | 31.97$^{(4)}$ | 32.20$^{(4)}$ | 30.93$^{(4)}$ | 29.14$^{(4)}$ | 29.03$^{(4)}$ | 29. 12$^{(4)}$ | 28.00$^{(4)}$ |
| EBOM | 87.10 | 50.32 | 39.93 | 34.07 | 34.77 | 31.54 | 33.72 | 30.86 | 30.87 | 32.38 | 30.18 | 29.67 | 29.26 | 29.07 | 28.66 |
| EPSM | 31.85 | 35.47 | 33.62 | 31.07 | 33.49 | 32.36 | 34.42 | 31.90 | 31.06 | 31.66 | 30.35 | 29.57 | 29.02 | 29.38 | 28.92 |
| EPSMA | **27.73** | **28.83** | 30.62 | **27.60** | **28.29** | 28.63 | **30.34** | **27.97** | **28.12** | **29.52** | **28.33** | **27.22** | **26.18** | **26.90** | **25.45** |
| FSBNDM | 109.20 | 69.49 | 51 14 | 42 92 | 42 74 | 36.30 | 37.88 | 31.97 | 32.49 | 33.75 | 30.04$^{(W4)}$ | 29.55$^{(W4)}$ | 29 38$^{(W4)}$ | 30.03$^{(W4)}$ | 29.16$^{(W4)}$ |
| FSBNDMq | 108.95$^{(21)}$ | 60.35$^{(20)}$ | 43.24$^{(20)}$ | 36.53$^{(20)}$ | 36 35$^{(31)}$ | 31 78$^{(31)}$ | 33 07$^{(31)}$ | 30.12$^{(31)}$ | 30.67$^{(31)}$ | 31.42$^{(31)}$ | 29.21$^{(31)}$ | 28.96$^{(31)}$ | 29.01$^{(31)}$ | 29.09$^{(31)}$ | 28.35$^{(31)}$ |
| MEMCMP | 410.11 | 515.34 | 530.96 | 610.32 | 590.86 | 594.91 | 507.26 | 672.11 | 630.67 | 660.49 | 621.84 | 610.59 | 597.82 | 660.38 | 635.67 |
| N32-freq | 29.16$^{(2)}$ | 30.55$^{(3)}$ | **29.44**$^{(2)}$ | 28.05$^{(3)}$ | 30.10$^{(3)}$ | 28.41$^{(3)}$ | 31.34$^{(3)}$ | 29.05$^{(3)}$ | 29.97$^{(3)}$ | 31.46$^{(3)}$ | 29.35$^{(3)}$ | 28.62$^{(3)}$ | 28.48$^{(3)}$ | 29.23$^{(3)}$ | 28.13$^{(3)}$ |
| N32-fixed | 28.18$^{(2)}$ | 30.46$^{(3)}$ | 29.51$^{(5)}$ | 28.26$^{(3)}$ | 30.36$^{(3)}$ | **28.28**$^{(3)}$ | 31.48$^{(3)}$ | 29.55$^{(3)}$ | 30.18$^{(3)}$ | 31.37$^{(3)}$ | 29.52$^{(3)}$ | 28.88$^{(3)}$ | 28.63$^{(3)}$ | 29.43$^{(3)}$ | 28.02$^{(3)}$ |
| SBNDMq | 115 40$^{(2)}$ | 60.38$^{(2)}$ | 44.42$^{(2)}$ | 36.60$^{(2)}$ | 36.91$^{(2)}$ | 32.62$^{(2)}$ | 34.39$^{(2)}$ | 30.25$^{(2)}$ | 30.85$^{(4)}$ | 31.59$^{(4)}$ | 29.40$^{(4)}$ | 28.48$^{(2)}$ | 28.41$^{(4)}$ | 29.20$^{(4)}$ | 28.18$^{(4)}$ |
| SKIPq | 108 73$^{(2)}$ | 65.88$^{(2)}$ | 53.10$^{(2)}$ | 43.19$^{(2)}$ | 41.55$^{(4)}$ | 34.60$^{(4)}$ | 34.85$^{(4)}$ | 30.80$^{(4)}$ | 30.89$^{(4)}$ | 31.82$^{(4)}$ | 29.76$^{(4)}$ | 28.77$^{(4)}$ | 28.31$^{(4)}$ | 28.88$^{(4)}$ | 27.52$^{(4)}$ |
| SSEF | – | – | – | – | – | – | – | – | – | – | 58.49 | 57.52 | 35.81 | 36.60 | 35.41 |
| TVSBS-W8 | 161.27 | 98.60 | 71.25 | 54.96 | 54.79 | 45.02 | 44.48 | 35.91 | 34.51 | 35.95 | 31.28 | 29.98 | 29.73 | 29.16 | 28.09 |
| UFNDMq | 193 73$^{(2)}$ | 97.48$^{(2)}$ | 74.22$^{(2)}$ | 59.57$^{(2)}$ | 54.45$^{(2)}$ | 46.19$^{(2)}$ | 44.47$^{(2)}$ | 35.54$^{(2)}$ | 34.01$^{(2)}$ | 34. 10$^{(2)}$ | 31.41$^{(2)}$ | 30.46$^{(2)}$ | 29.94$^{(2)}$ | 28.73$^{(2)}$ | 27.39$^{(2)}$ |
| WFRq | 133 91$^{(2)}$ | 75.74$^{(2)}$ | 54.29$^{(3)}$ | 46.53$^{(2)}$ | 43.13$^{(3)}$ | 34.54$^{(3)}$ | 35.92$^{(3)}$ | 30.43$^{(4)}$ | 30.98$^{(4)}$ | 32.02$^{(4)}$ | 29.74$^{(4)}$ | 29.02$^{(5)}$ | 28.83$^{(4)}$ | 28.78$^{(4)}$ | 27.80$^{(4)}$ |

**Table 4.4** : Running times for English text when pattern length < 64.

| m | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 28 | 32 | 40 | 48 | 56 | 62 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BNDMq | 129 68$^{(2)}$ | 70 10$^{(2)}$ | 56 46$^{(2)}$ | 55 19$^{(2)}$ | 47 51$^{(4)}$ | 40 54$^{(4)}$ | 36 30$^{(4)}$ | 34 10$^{(4)}$ | 32 65$^{(4)}$ | 31 13$^{(4)}$ | 30 34$^{(4)}$ | 29 26$^{(4)}$ | 29 07$^{(4)}$ | 28 70$^{(4)}$ | 28 58$^{(4)}$ |
| BOM2 | | 151.19 | 134.14 | 127.07 | 110.93 | 90.81 | 82.06 | 65.94 | 58.65 | 50.18 | 47.19 | 45.21 | 43.17 | 44.63 | 43.27 |
| BSDMq | 103.97$^{(2)}$ | 60.44$^{(2)}$ | 50.90$^{(2)}$ | 46.71$^{(3)}$ | 41.73$^{(3)}$ | 37.74$^{(4)}$ | 34.52$^{(4)}$ | 32.76$^{(5)}$ | 31.98$^{(6)}$ | 31.80$^{(6)}$ | 31.44$^{(6)}$ | 30.05$^{(6)}$ | 29.10$^{(6)}$ | 28.58$^{(6)}$ | 28.64$^{(6)}$ |
| EBOM | 100.05 | 60.69 | 51.36 | 50.68 | 46.87 | 44.56 | 43.42 | 42.27 | 41.21 | 40.23 | 39.06 | 36.77 | 38.90 | 36.30 | 34.05 |
| EPSM | 31.48 | 32.54 | 36.43 | 41.52 | 38.80 | 38.13 | 32.74 | 32.67 | 31.12 | 30.56 | 28.93 | 28.01 | 27.83 | 27.07 | 26.87 |
| EPSMA | **26.94** | **27.37** | 31.81 | 31.67 | **29.73** | **29.77** | **28.70** | **28.42** | **27.03** | **26.67** | **25.80** | **25.09** | **24.71** | **24.13** | **23.26** |
| FSBNDM | 130.47 | 83 09 | 68.00 | 63.11 | 57.47$^{(W4)}$ | 49.01$^{(W4)}$ | 46.36$^{(W4)}$ | 40.45$^{(W4)}$ | 38.68$^{(W4)}$ | 36.35$^{(W2)}$ | 34.56$^{(W4)}$ | 32.34$^{(W4)}$ | 31.57$^{(W4)}$ | 31.20$^{(W4)}$ | 30.92$^{(W4)}$ |
| FSBNDMq | 127.81$^{(20)}$ | 67.39$^{(20)}$ | 51.61$^{(31)}$ | 47.22$^{(31)}$ | 41.75$^{(31)}$ | 38 96$^{(31)}$ | 35.34$^{(41)}$ | 33.40$^{(41)}$ | 32 08$^{(41)}$ | 31.43$^{(41)}$ | 30.88$^{(41)}$ | 29.46$^{(31)}$ | 29.08$^{(41)}$ | 28.75$^{(41)}$ | 28.47$^{(41)}$ |
| MEMCMP | 430.84 | 476.10 | 560.84 | 540.93 | 600.90 | 600.76 | 621.13 | 640.32 | 640.29 | 670.72 | 678.62 | 680.78 | 710.80 | 680.38 | 670.58 |
| N32-freq | 30.09$^{(2)}$ | 29.89$^{(3)}$ | **29.87**$^{(3)}$ | **30.73**$^{(3)}$ | 31.03$^{(2)}$ | 30.36$^{(2)}$ | 30.20$^{(2)}$ | 30.07$^{(2)}$ | 29.17$^{(3)}$ | 29.10$^{(5)}$ | 28.89$^{(3)}$ | 28.48$^{(2)}$ | 28.03$^{(3)}$ | 28.01$^{(3)}$ | 27.10$^{(2)}$ |
| N32-fixed | 29.95$^{(2)}$ | 29.60$^{(3)}$ | 30.16$^{(3)}$ | 30.78$^{(3)}$ | 31.26$^{(3)}$ | 30.62$^{(3)}$ | 30.22$^{(3)}$ | 30.38$^{(3)}$ | 29.45$^{(3)}$ | 29.07$^{(3)}$ | 29.03$^{(3)}$ | 28.56$^{(3)}$ | 28.12$^{(3)}$ | 28.13$^{(3)}$ | 27.15$^{(3)}$ |
| SBNDMq | 122.62$^{(2)}$ | 67.31$^{(2)}$ | 54.35$^{(2)}$ | 52.68$^{(2)}$ | 47.50$^{(4)}$ | 44.20$^{(4)}$ | 36.48$^{(4)}$ | 34.39$^{(4)}$ | 32.82$^{(4)}$ | 31.17$^{(4)}$ | 30.26$^{(4)}$ | 29.83$^{(4)}$ | 29.50$^{(4)}$ | 29.01$^{(4)}$ | 28.55$^{(4)}$ |
| SKIPq | 112.14$^{(2)}$ | 65.86$^{(2)}$ | 53.06$^{(2)}$ | 52.70$^{(2)}$ | 44.60$^{(4)}$ | 39.43$^{(4)}$ | 35.36$^{(4)}$ | 33.31$^{(4)}$ | 32.98$^{(4)}$ | 31.55$^{(4)}$ | 31.20$^{(4)}$ | 30.35$^{(4)}$ | 29.44$^{(4)}$ | 28.95$^{(4)}$ | 28.50$^{(4)}$ |
| SSEF | – | – | – | – | – | – | – | – | – | – | 58.44 | 55.68 | 40.92 | 36.92 | 35.27 |
| TVSBS-W8 | 169.07 | 98.88 | 71.21 | 64.94 | 56.84 | 48.18 | 45.17 | 39.50 | 38.21 | 36.78 | 35.68 | 35.57 | 38.41 | 37.22 | 36.80 |
| UFNDMq | 118.71$^{(2)}$ | 178 53$^{(2)}$ | 66.67$^{(2)}$ | 68.07$^{(2)}$ | 58.51$^{(8)}$ | 45.96$^{(2)}$ | 43.77$^{(8)}$ | 38.25$^{(8)}$ | 35.43$^{(8)}$ | 32.34$^{(8)}$ | 32.02$^{(8)}$ | 31.86$^{(8)}$ | 32.11$^{(8)}$ | 33.15$^{(8)}$ | 32.40$^{(8)}$ |
| WFRq | 139.43$^{(2)}$ | 77.38$^{(2)}$ | 58.99$^{(3)}$ | 57.83$^{(2)}$ | 46.27$^{(3)}$ | 39.25$^{(3)}$ | 36.13$^{(4)}$ | 33.58$^{(4)}$ | 31.44$^{(3)}$ | 31.08$^{(4)}$ | 30.61$^{(4)}$ | 29.20$^{(4)}$ | 28.85$^{(4)}$ | 28.51$^{(4)}$ | 28.10$^{(4)}$ |

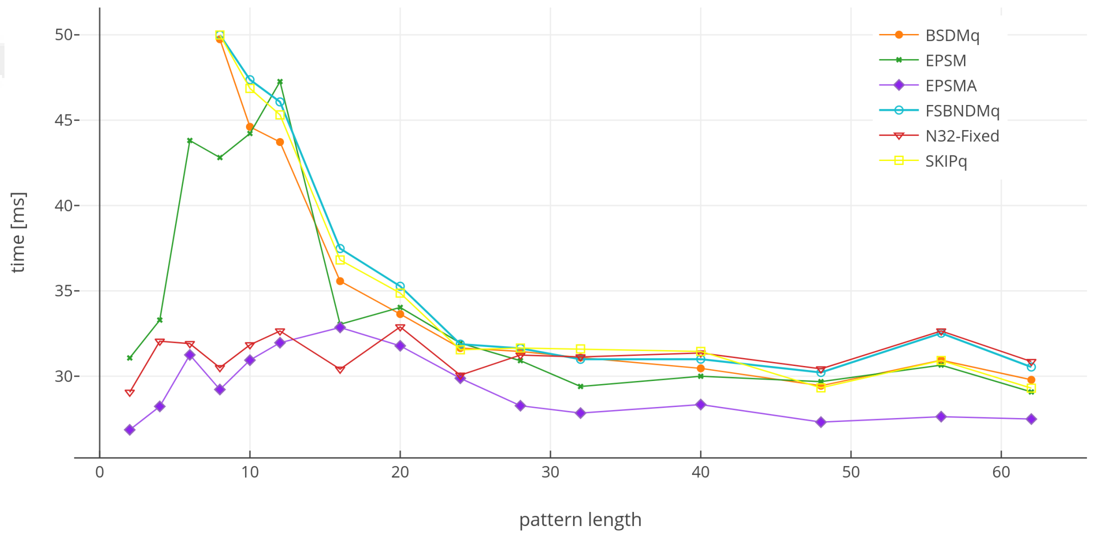


**Figure 4.1** : Times for genome sequence, pattern length<64.

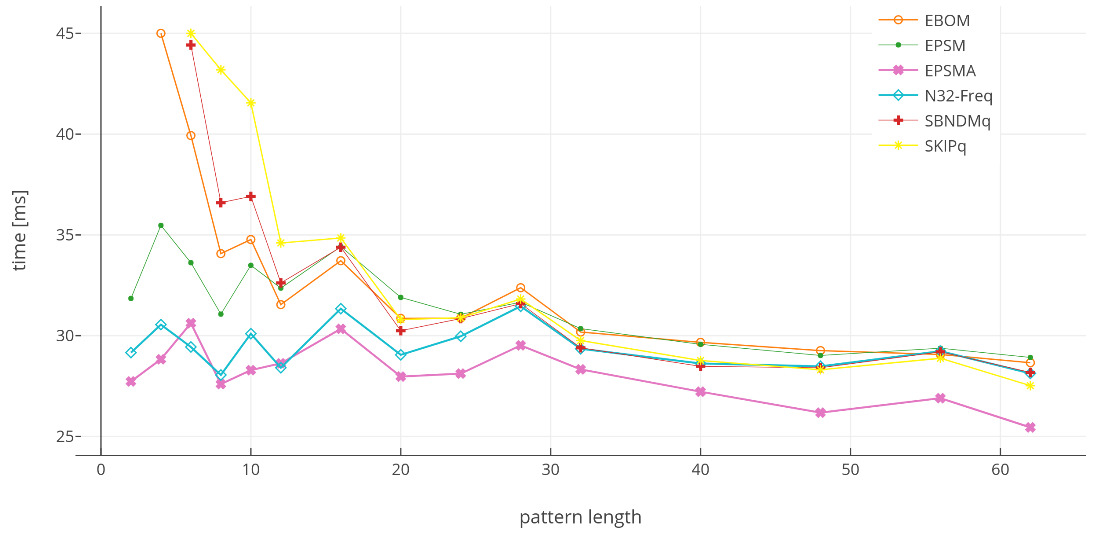


**Figure 4.2** : Times for protein sequence, pattern length<64.

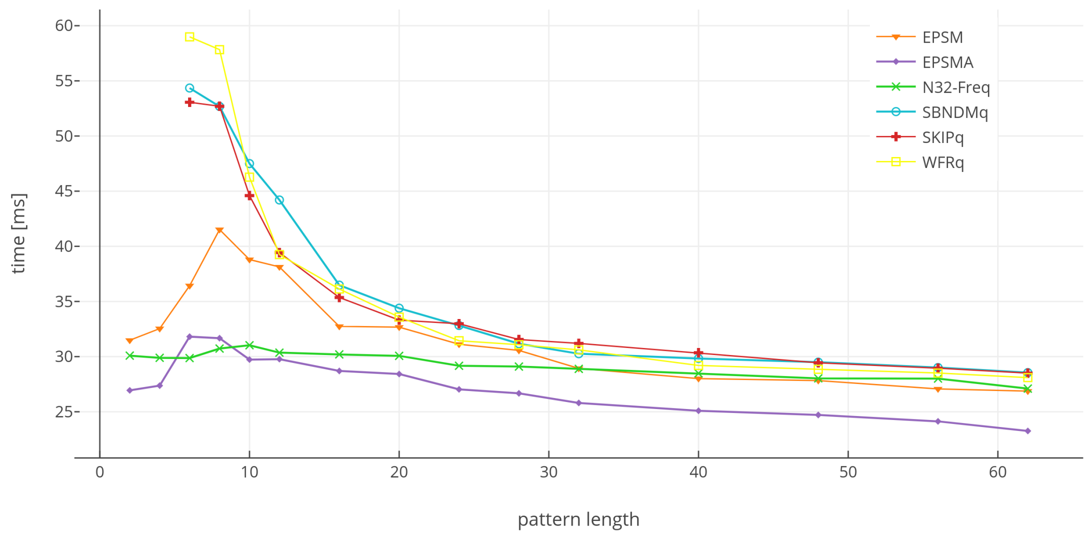


**Figure 4.3** : Times for English text, pattern length<64.

Time values are also represented in the above figures for more apprehensible interpretation of results. It can be seen from the figures, EPSMA is faster than almost all previous most competitive exact string matching algorithms in the literature.

## 4.2 Results for SSEFA Algorithm (patlen>64)

SSEFA-1 and SSEFA-2, non-optimized versions of SSEFA, are added in algorithm test list for comparison between optimized SSEFA. Hereby, it can be seen how cache optimization can effect the overall performance.

**Table 4.5** : Running times for genome sequence when pattern length > 64.

| **m** | **64** | **96** | **128** | **192** | **256** | **384** | **512** | **A768** | **1024** | **1280** | **1536** | **1792** | **2048** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BNDMq | 31.67(6) | 30.46(6) | 29.70(6) | 29.97(6) | 29.56(6) | 29 80(6) | 32.07(6) | 29.87(6) | 29.91(6) | 31.06(6) | 31.00(6) | 31.09(6) | 28.92(6) |
| BOM2 | 51.89 | 39.45 | 38.04 | 35.56 | 29.67 | 29.48 | 28.10 | 24.53 | 22.82 | 22.94 | 21.83 | 21.92 | 20.77 |
| BSDMq | 30.22(8) | 29.80(8) | 29.82(8) | 30.10(7) | 27.30(8) | 27 27(8) | 26.84(7) | 27.15(7) | 27.25(7) | 28.47(7) | 27.61(7) | 28.26(6) | 27.17(7) |
| EBOM | 47.15 | 36.86 | 35.97 | 34.79 | 31.45 | 29.13 | 28.76 | 24.14 | 22.37 | 22.63 | 21.59 | 21.55 | 20.55 |
| EPSM | 30.33 | 29.23 | 28.11 | 26.47 | 23.00 | 21.01 | 20.01 | 18.13 | 17.25 | 18.26 | 17.72 | 17.90 | 17.04 |
| FSBNDM-Wq | 48.36(2) | 43.80(2) | 48.49(2) | 44.24(2) | 43.39(2) | 43 17(2) | 47.80(2) | 43.57(2) | 43.59(2) | 44.18(2) | 47.60(2) | 44.22(2) | 42.25(2) |
| FSBNDMq | 31.75(61) | 30.31(61) | 31.77(61) | 29.86(61) | 30.04(61) | 29.71(61) | 30.02(61) | 29.84(61) | 29 73(61) | 30.85(61) | 30.91(61) | 30.88(61) | 29.75(61) |
| MEMCMP | 780.89 | 900.51 | 780.82 | 840.74 | 830.13 | 752.03 | 732.38 | 741.38 | 720.72 | 708.84 | 760.97 | 820.90 | 850.66 |
| N32-freq | 32.33(5) | 31.01(5) | 32.52(5) | 30.44(5) | 30.09(5) | – | – | – | – | – | – | – | – |
| N32-fixed | 32.05(5) | 30.70(5) | 32.20(5) | 30.86(5) | 29.22(5) | – | – | – | – | – | – | – | – |
| SBNDMq | 31.93(6) | 30.54(6) | 32.11(6) | 30.10(6) | 29.73(6) | 28 93(6) | 32.23(6) | 29.97(6) | 28.90(6) | 31.13(6) | 30.88(6) | 31.03(6) | 29.98(6) |
| SKIPq | 30.99(7) | 28.30(6) | 29.50(6) | 27.94(6) | 26.34(7) | 22 20(8) | 21.73(8) | 18.27(8) | 17.03(8) | 18.05(8) | 17.58(8) | 17.74(8) | 16.85(8) |
| SSEF | 33.93 | 29.89 | 29.16 | 26.78 | 24.53 | 20.01 | 18.48 | 17.66 | 16.35 | 16.23 | 16.06 | 15.81 | 15.24 |
| SSEFA | **28.26** | **27.64** | **25.52** | **23.76** | **20.66** | **17.81** | **15.66** | **14.29** | **12.97** | **13.78** | **13.62** | **12.93** | **12.78** |
| SSEFA-1 | 130.24 | 85.70 | 57.78 | 52.00 | 39.49 | 29.29 | 28.77 | 26.51 | 28.22 | 25.31 | 23.81 | 30.92 | 24.71 |
| SSEFA-2 | 70.64 | 46.37 | 39.75 | 34.28 | 30.87 | 23.53 | 21.24 | 19.72 | 23.74 | 19.28 | 19.02 | 22.53 | 19.49 |
| TVSBS-W8 | 91.12 | 77.24 | 78.45 | 75.81 | 74.20 | 75.80 | 85.59 | 76.15 | 76.94 | 78.25 | 86.23 | 79.98 | 78.61 |
| UFNDMq | 36.49(8) | 34.38(8) | 33.74(8) | 33.88(8) | 33.48(8) | 33 81(8) | 35.71(8) | 33.87(8) | 33.76(8) | 34.86(8) | 35.64(8) | 34.79(8) | 32.82(8) |
| WFRq | 30.84(5) | 28.75(5) | 28.65(6) | 27.00(2) | 25.12(5) | 22 12(6) | 21.11(7) | 18.09(7) | 17.53(7) | 18.33(4) | 17.48(4) | 17.72(4) | 16.78(3) |

**Table 4.6** : Running times for protein sequence when pattern length > 64.

| m | 64 | 96 | 128 | 192 | 256 | 384 | 512 | 768 | 1024 | 1280 | 1536 | 1792 | 2048 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BNDMq | 30.10$^{(4)}$ | 31.16$^{(4)}$ | 30.21$^{(4)}$ | 30.32$^{(4)}$ | 30.33$^{(4)}$ | 30.34$^{(2)}$ | 30.25$^{(4)}$ | 30.25$^{(4)}$ | 29.27$^{(4)}$ | 30.69$^{(2)}$ | 30.36$^{(2)}$ | 29.19$^{(4)}$ | 29.31$^{(4)}$ |
| BOM2 | 31.48 | 31.37 | 30.45 | 30.07 | 28.01 | 25.09 | 22.57 | 21.30 | 18.97 | 19.06 | 20.21 | 19.46 | 19.89 |
| BSDMq | 29.95$^{(4)}$ | 30.68$^{(4)}$ | 29.45$^{(4)}$ | 29.08$^{(4)}$ | 29.35$^{(4)}$ | 28.77$^{(4)}$ | 29.06$^{(4)}$ | 28.64$^{(4)}$ | 27.58$^{(4)}$ | 29.24$^{(4)}$ | 28.56$^{(4)}$ | 27.52$^{(4)}$ | 27.48$^{(4)}$ |
| EBOM | 29.80 | 30.31 | 29.11 | 27.87 | 26.53 | 23.59 | 21.75 | 20.29 | 18.94 | 19.49 | 20.23 | 19.44 | 19.80 |
| EPSM | 29.43 | 29.86 | 29.01 | 26.02 | 23.55 | 21.03 | 18.59 | 18.77 | 17.69 | 18.12 | 18.65 | 17.71 | 17.67 |
| FSBNDM-Wq | 32.02$^{(4)}$ | 32.84$^{(4)}$ | 32.12$^{(4)}$ | 30.67$^{(4)}$ | 32.35$^{(4)}$ | 30.74$^{(4)}$ | 32.19$^{(4)}$ | 30.75$^{(4)}$ | 29.72$^{(4)}$ | 30.94$^{(4)}$ | 30.74$^{(4)}$ | 29.88$^{(4)}$ | 29.79$^{(4)}$ |
| FSBNDMq | 29.96$^{(31)}$ | 30 76$^{(31)}$ | 29.73$^{(31)}$ | 29.89$^{(31)}$ | 30.24$^{(31)}$ | 29.95$^{(31)}$ | 30.03$^{(31)}$ | 29.87$^{(31)}$ | 28.83$^{(31)}$ | 30.34$^{(31)}$ | 29.86$^{(31)}$ | 28.75$^{(31)}$ | 28.86$^{(31)}$ |
| MEMCMP | 660.50 | 590.94 | 601.14 | 596.23 | 600.87 | 587.45 | 602.25 | 607.47 | 591.54 | 613.91 | 608.38 | 620.32 | 590.18 |
| N32-freq | 30.41$^{(3)}$ | 31.29$^{(3)}$ | 30.31$^{(3)}$ | 30.12$^{(3)}$ | 30.66$^{(3)}$ | – | – | – | – | – | – | – | – |
| N32-fixed | 30.04$^{(3)}$ | 30.71$^{(3)}$ | 30.11$^{(3)}$ | 29.97$^{(3)}$ | 30.10$^{(3)}$ | – | – | – | – | – | – | – | – |
| SBNDMq | 30.38$^{(4)}$ | 31.08$^{(4)}$ | 30.22$^{(4)}$ | 30.24$^{(4)}$ | 30.40$^{(4)}$ | 30.22$^{(4)}$ | 30.31$^{(4)}$ | 30.28$^{(4)}$ | 29.17$^{(4)}$ | 30.42$^{(4)}$ | 30.26$^{(4)}$ | 29.08$^{(2)}$ | 29.19$^{(4)}$ |
| SKIPq | 29.79$^{(4)}$ | 30.05$^{(4)}$ | 28.28$^{(4)}$ | 27.33$^{(4)}$ | 25.86$^{(4)}$ | 22.08$^{(4)}$ | 19.78$^{(8)}$ | 18.41$^{(8)}$ | 17.11$^{(8)}$ | 17.48$^{(8)}$ | 18.00$^{(8)}$ | 17.04$^{(8)}$ | 16.97$^{(8)}$ |
| SSEF | 34.18 | 32.52 | 30.28 | 27.14 | 23.13 | 20.17 | 18.62 | 17.26 | 17.03 | 16.02 | 16.94 | 16.07 | 15.87 |
| SSEFA | **28.16** | 28.95 | **27.45** | **24.88** | **21.12** | **17.21** | **15.40** | **15.01** | **14.53** | **13.90** | **14.06** | **13.56** | **12.56** |
| SSEFA-1 | 88.27 | 67.20 | 48.37 | 38.91 | 34.82 | 28.84 | 26.73 | 25.28 | 24.96 | 24.50 | 23.90 | 24.69 | 23.78 |
| SSEFA-2 | 69.31 | 43.81 | 36.56 | 30.81 | 27.24 | 23.04 | 19.76 | 18.77 | 18.33 | 18.32 | 18.78 | 19.62 | 18.32 |
| TVSBS-W8 | 30.10 | **28.16** | 27.75 | 26.69 | 27.36 | 26.97 | 26.71 | 25.01 | 23.14 | 24.31 | 23.82 | 23.38 | 22.55 |
| UFNDMq | 30.03$^{(2)}$ | 29.76$^{(2)}$ | 27.79$^{(2)}$ | 25.89$^{(2)}$ | 29.28$^{(2)}$ | 28.41$^{(2)}$ | 26.45$^{(2)}$ | 26.44$^{(2)}$ | 23.42$^{(2)}$ | 25.50$^{(2)}$ | 23.59$^{(2)}$ | 22.38$^{(2)}$ | 23.86$^{(2)}$ |
| WFRq | 29.59$^{(5)}$ | 29.71$^{(5)}$ | 28.18$^{(5)}$ | 27.29$^{(4)}$ | 25.37$^{(4)}$ | 22.39$^{(4)}$ | 20.33$^{(5)}$ | 19.25$^{(5)}$ | 17.25$^{(3)}$ | 17.53$^{(3)}$ | 17.80$^{(3)}$ | 17.21$^{(6)}$ | 16.92$^{(3)}$ |

**Table 4.7** : Running times for English text when pattern length > 64.

| m | 64 | 96 | 128 | 192 | 256 | 384 | 512 | 768 | 1024 | 1280 | 1536 | 1792 | 2048 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BNDMq | 31.76$^{(2)}$ | 30.13$^{(4)}$ | 31.75$^{(6)}$ | 33.46$^{(4)}$ | 29.21$^{(4)}$ | 29.99$^{(4)}$ | 31.62$^{(4)}$ | 31.41$^{(4)}$ | 31.77$^{(4)}$ | 33.15$^{(4)}$ | 32.84$^{(4)}$ | 29.63$^{(4)}$ | 31.38$^{(4)}$ |
| BOM2 | 41.34 | 33.92 | 33.72 | 33.89 | 28.44 | 28.74 | 29.18 | 26.45 | 24.84 | 24.68 | 24.45 | 21.17 | 22.78 |
| BSDMq | 30.46$^{(6)}$ | 28.44$^{(6)}$ | 29.73$^{(8)}$ | 31.08$^{(6)}$ | 27.17$^{(7)}$ | 27.37$^{(6)}$ | 28.95$^{(6)}$ | 28.61$^{(6)}$ | 29.09$^{(6)}$ | 30.06$^{(6)}$ | 29.84$^{(6)}$ | 26.42$^{(6)}$ | 28.44$^{(8)}$ |
| EBOM | 36.72 | 33.08 | 32.64 | 33.93 | 28.28 | 27.55 | 27.52 | 24.79 | 23.83 | 24.16 | 23.69 | 20.57 | 22.12 |
| EPSM | 29.72 | 29.19 | 28.85 | 26.07 | 22.02 | 19.57 | 18.70 | 18.83 | 18.76 | 18.86 | 19.68 | 17.92 | 18.77 |
| FSBNDM-Wq | 37.41$^{(4)}$ | 33.69$^{(4)}$ | 37.17$^{(4)}$ | 38.56$^{(4)}$ | 33.17$^{(4)}$ | 33.92$^{(4)}$ | 37.36$^{(4)}$ | 35.09$^{(4)}$ | 37.05$^{(4)}$ | 38.24$^{(4)}$ | 38.17$^{(4)}$ | 33.66$^{(4)}$ | 35.12$^{(4)}$ |
| FSBNDMq | 31.58$^{(41)}$ | 30 90$^{(41)}$ | 31.41$^{(41)}$ | 33.32$^{(41)}$ | 29.07$^{(41)}$ | 29.69$^{(41)}$ | 31.60$^{(41)}$ | 31.24$^{(41)}$ | 31.73$^{(41)}$ | 32.90$^{(41)}$ | 32.69$^{(41)}$ | 29.43$^{(41)}$ | 31.24$^{(41)}$ |
| MEMCMP | 760.81 | 711.78 | 632.58 | 600.81 | 670.06 | 633.48 | 570.78 | 580.52 | 592.52 | 612.76 | 602.17 | 610.84 | 612.23 |
| N32-freq | 30.49$^{(2)}$ | 29.28$^{(2)}$ | 29.56$^{(2)}$ | 31.29$^{(2)}$ | 27.30$^{(2)}$ | – | – | – | – | – | – | – | – |
| N32-fixed | 30.89$^{(3)}$ | 28.81$^{(3)}$ | 30.20$^{(3)}$ | 31.83$^{(3)}$ | 27.88$^{(3)}$ | – | – | – | – | – | – | – | – |
| SBNDMq | 31.69$^{(4)}$ | 30.01$^{(4)}$ | 31.51$^{(4)}$ | 33.31$^{(4)}$ | 29.12$^{(4)}$ | 29.83$^{(4)}$ | 31.51$^{(4)}$ | 31.33$^{(4)}$ | 31.81$^{(4)}$ | 33.03$^{(4)}$ | 32.99$^{(4)}$ | 29.53$^{(4)}$ | 31.30$^{(4)}$ |
| SKIPq | 30.57$^{(6)}$ | 28.91$^{(4)}$ | 29.05$^{(5)}$ | 30.20$^{(4)}$ | 24.49$^{(8)}$ | 20.80$^{(8)}$ | 19.05$^{(8)}$ | 17.53$^{(8)}$ | 17.31$^{(8)}$ | 18.36$^{(8)}$ | 18.14$^{(8)}$ | 15.34$^{(8)}$ | 17.10$^{(8)}$ |
| SSEF | 33.63 | 31.78 | 30.95 | 28.49 | 21.30 | 19.13 | 17.37 | 16.20 | 16.14 | 17.19 | 17.12 | 16.44 | 16.00 |
| SSEFA | **27.94** | 27.58 | **27.39** | **25.67** | **19.04** | **15.90** | **15.67** | **14.63** | **14.41** | **14.92** | **15.10** | **13.75** | **13.43** |
| SSEFA-1 | 78.04 | 56.75 | 54.05 | 38.50 | 33.89 | 36.32 | 26.09 | 25.40 | 24.33 | 26.42 | 25.62 | 23.81 | 22.87 |
| SSEFA-2 | 61.29 | 45.34 | 38.89 | 30.99 | 26.31 | 28.68 | 19.15 | 19.04 | 18.47 | 19.36 | 19.58 | 18.19 | 17.83 |
| TVSBS-W8 | 31.99 | 28.08 | 28.93 | 32.73 | 26.55 | 27.60 | 28.24 | 25.74 | 25.20 | 25.10 | 24.19 | 21.57 | 22.41 |
| UFNDMq | 33.96$^{(2)}$ | 28.23$^{(2)}$ | 30.23$^{(4)}$ | 35.76$^{(2)}$ | 26.43$^{(2)}$ | 26.31$^{(2)}$ | 27.60$^{(4)}$ | 28.36$^{(2)}$ | 23.23$^{(2)}$ | 24.90$^{(2)}$ | 24.81$^{(2)}$ | 22.34$^{(2)}$ | 23.43$^{(2)}$ |
| WFRq | 30.23$^{(4)}$ | **27.40**$^{(4)}$ | 28.56$^{(4)}$ | 29.70$^{(4)}$ | 23.59$^{(4)}$ | 21.67$^{(4)}$ | 20.92$^{(5)}$ | 19.38$^{(5)}$ | 18.94$^{(3)}$ | 19.75$^{(3)}$ | 19.60$^{(3)}$ | 16.55$^{(3)}$ | 18.16$^{(3)}$ |

The SSEFA (optimized version) algorithm is compared with the following algorithms in addition to the previously mentioned algorithms in EPSMA on SMART Tool.
- SSEF: filter based fast matching by using SSE instructions [9];
- SSEFA-1: non-optimized version of SSEFA with unmasked 32-bit filter;
- SSEFA-2: reduced filter length version of SSEFA with single 28-bit filter;
Time values are depicted graphically show that SSEFA algorithm gives better results for most conditions than other algorithms which are known as fastest in the literature.

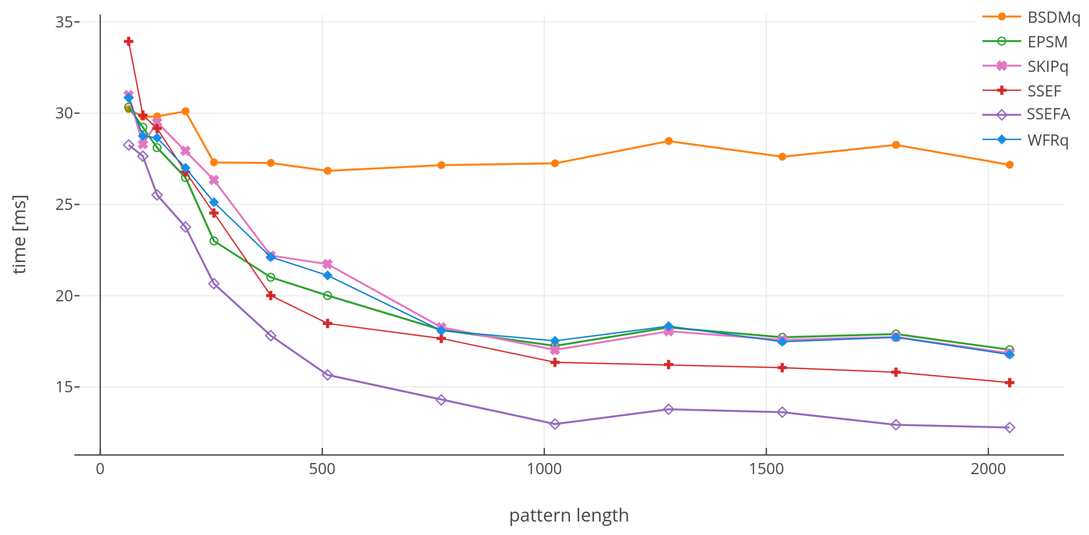


**Figure 4.4** : Times for genome sequence, pattern length>64.

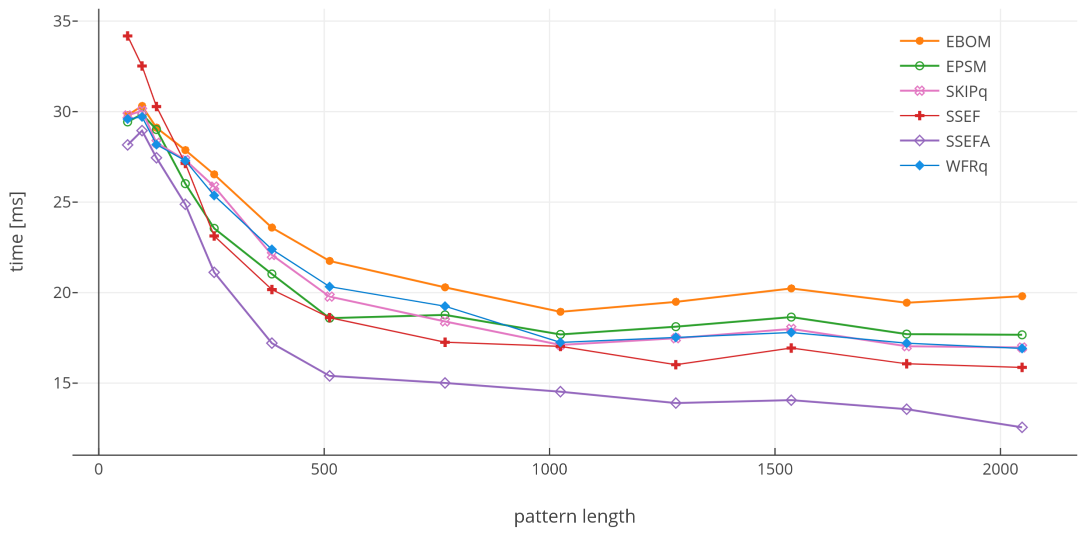


**Figure 4.5** : Times for protein sequence, pattern length>64.

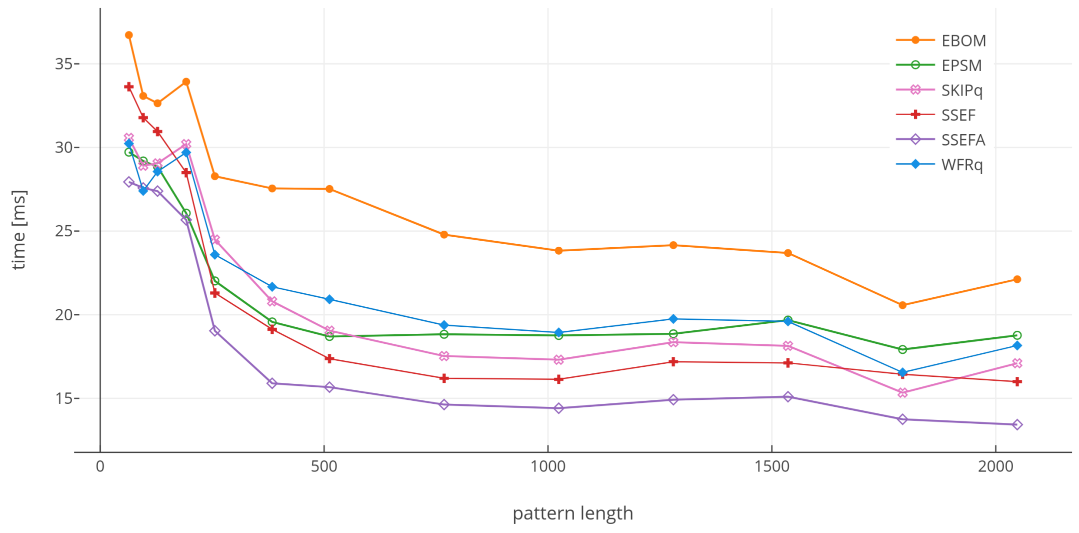


**Figure 4.6** : Times for English text, pattern length>64.

# 5. CONCLUSIONS AND RECOMMENDATIONS

The new state-of-the-art variations of the EPSM and SSEF algorithms are proposed by adding new techniques and instructions for the exact string matching problem. AVX2 instructions are utilized and optimizations are applied using analyzes on Intel Vtune Amplifier to improve overall performance. Besides using AVX2 instructions operated on 256-bit data for filtering algorithms, optimal bit length of the filter, 2-stage filtering technique and data structure for storing the values of filters have the significant impacts on performance. Experiments show that new algorithms are faster than almost all previous most efficient exact string matching algorithms for various pattern lengths and alphabet sizes. By the way, EPSMA and SSEFA don't have the best result at a few pattern lengths, these lengths are 6, 8, 96 for English text; 6, 12, 128 for protein sequence and 16 for genome sequence. Time difference between EPSMA and EPSM algorithms becomes more evident when pattern length is smaller than 16. Likewise, EPSMA gives better results against to the nearest competitors when pattern length is greater than 16 but the time difference between EPSMA and others is more stable in this range. Fine speedups are achieved for SSEFA over old version while changing the pattern length and text types. Particularly time difference becomes more well-marked between SSEFA and other algorithms for very long patterns. Eventually, proposed algorithms are extremely useful for practitioners.

As a future work, new algorithms may be implemented using equivalent SIMD instructions with some modifications on other architectures like the ARM, AMD. Furthermore, new efficient word-size instructions can be defined utilizing AVX2 intrinsics for other types of string matching such as approximate, circular matching and other issues related to string.

## REFERENCES

[1] **Faro, S.** (2016). Exact online string matching bibliography, *arXiv preprint arXiv:1605.05067*.

[2] **D. E. Knuth, J. H. Morris, J. and Pratt., V.R.** (1977). Fast pattern matching in strings, *6(2)*, 323–350.

[3] **Boyer, R.S. and Moore, J.S.** (1977). A fast string searching algorithm, *Communication of the ACM*, *20*(10), 762–772.

[4] **B. Durian, J. Holub, H.P. and Tarhio, J.** (2009). Tuning bndm with q-grams. Proceedings of the Workshop on Algorithm Engineering and Experiments, *ALENEX*, pp.29–37.

[5] **Karp, R.M. and Rabin, M.O.** (1987). Efficient randomized pattern-matching algorithms, *IBM Journal of Research and Development Mathematics and computing*, *31(2)*, 249–260.

[6] **Faro, S. and Lecroq, T.** (2013). The exact online string matching problem: A review of the most recent results, *ACM Computing Surveys (CSUR)*, *45*(2).

[7] **Faro, S. and Lecroq, T.** (2010). The exact string matching problem: a comprehensive experimental evaluation, *arXiv preprint arXiv:1012.2547*.

[8] **S. Faro, M.K.** (2013). Fast packed string matching for short patterns, *15th Meeting on Algorithm Engineering and Experiments, ALENEX*, SIAM, New Orleans, LA, USA, pp.113–121.

[9] **KULEKCI, M.O.** (2009). Filter based fast matching of long patterns by using simd instructions, *Prague Stringology Conference*, Czech Technical University in Prague, Czech Republic, pp.118–128.

[10] **Tamanna Chhabra, Simone Faro, M.O.K. and Tarhio, J.** (2017). Engineering order-preserving pattern matching with SIMD parallelism, *Software: Practice and Experience*, *47*, 731–739.

[11] **Faro, S. and Kulekci, M.O.** (2012). Fast multiple string matching using streaming SIMD extensions technology, *19th International Symposium on String Processing and Information Retrieval*, volume7608 of LNCS, pp.217–228.

[12] **S. Ladra, O. Pedreira, J.D.N.B.** (2012). Exploiting SIMD instructions in current processors to improve classical string algorithms, *The 16th East European Conference on Advances in Databases and Information Systems*, volume7503 of LNCS, Springer, pp.254–267.

[13] **J. Tarhio, J.H. and Giaquinta, E.** (2017). Technology beats algorithms (in exact string matching, *Software: Practice and Experience*, *47(12)*, 1877–1885.

[14] **Flynn, M.J.** (1966). Very High-Speed Computing Systems, *Proceedings of the IEEE*, *54*(12), 1901 – 1909, `http://www.cs.utexas.edu/users/dburger/teaching/cs395t-s08/papers/5_flynn.pdf`.

[15] **Mutlu, O.**, Computer Architecture Lecture Notes, `https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=onur-comparch-fall2017-lecture8-afterlecture.pdf`.

[16] Introduction to Intel® Advanced Vector Extensions, `https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf`.

[17] Intel 64 and IA-32 Architectures Software Developer's Manual, `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`.

[18] Intel VTune Amplifier 2019 User Guide, `https://software.intel.com/en-us/=vtune-amplifier-help-analyze-performance`.

[19] Cache Miss Rates in Intel® VTune™ Amplifier
, `https://software.intel.com/en-us/articles/cache-miss-rates-in-intel-vtune-amplifier-xe`.

[20] **B. Durian, J. Holub, H.P. and Tarhio, J.** (2010). Improving practical exact string matching, *Information Processing Letters 110(4)*, 148–152.

[21] **S. Faro, T. Lecroq, S.B.S.D.M.A.M.** (2016). The String Matching Algorithms Research Tool, *Prague Stringology Conference*, volume7503 of LNCS, Springer, Prag, Czech Republic, pp.99–111.

[22] **McCaffrey, R. and Abers, G.** (2000). G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata, *ACM Journal of Experimental Algorithmics (JEA)*, *5(4)*.

[23] **Faro S., L.T.** (2012). A fast suffix automata based algorithm for exact online string matching, *Implementation and Application of Automata. CIAA*, volume7381 of LNCS, Springer, Berlin, Heidelberg.

[24] **Faro, S. and Lecroq, T.** (2008). Efficient variants of the backward-oracle-matching algorithm, *Prague Stringology Conference*, Springer, Prag, Czech Republic, pp.146–160.

[25] **Peltola, H. and Tarhio, J.** (2011). Variations of forward-sbndm, *Prague Stringology Conference*, Prag, Czech Republic.

[26] **Faro, S.** (2016). A very fast string matching algorithm based on condensed alphabets, *Algorithmic Aspects in Information and Management - 10th International Conference, AAIM*.

[27] **R. Thathoo, A. Virmani, S.L.N.B. and Sekar, K.** (2006). TVSBS: A fast exact pattern matching algorithm for biological sequences, *Current Science*, *91*(1), 47–53.

[28] **Domenico Cantone, Simone Faro, A.P.** (2017). Speeding Up String Matching by Weak Factor Recognition, *Stringology*, pp.242–50.

## CURRICULUM VITAE

**Mehmet Akif Aydoğmuş**

**Place and Date of Birth:** Eskişehir, 29.03.1989

**E-Mail:** aydogmusm@itu.edu.tr

**EDUCATION:**

- **B.Sc.:** 2012, Istanbul Technical University, Electronics Engineering

**PROFESSIONAL EXPERIENCE AND REWARDS:**

- 03-07.2012 Baykar Makina / R&D Engineer (Part Time)
- 2012-2013 AGENA BST / Software Development Engineer
- 2013-2015 NETAŞ / Software Development Engineer
- 2015- TÜBİTAK / Embedded Software Engineer - Researcher

**PUBLICATIONS, PRESENTATIONS AND PATENTS ON THE THESIS:**

- M. Akif Aydoğmuş and M. Oğuzhan Külekci, Optimizing Packed String Matching on AVX2 Platform, VECPAR 2018, 13th International Meeting on High Performance Computing for Computational Science (São Paulo, Brazil).

**OTHER PUBLICATIONS, PRESENTATIONS AND PATENTS:**

- AYDOGMUS, M. A., KUNTMAN, H. New CMOS realization of ZC-CG-CDBA and its filter application. In Proceedings of 20th. IEEE Signal Processing And Communications Applications Conference (SIU 2012). Muğla (Turkey), 2012, p. 18 - 20.