Exploit Every Cycle: Vectorized Time Series Algorithms on Modern Commodity CPUs

Bo Tang[1] Man Lung Yiu[1] Yuhong Li[2] Leong Hou U[2]

[1] Hong Kong Polytechnic University, Hung Hum, Hong Kong
{csbtang,csmlyiu}@comp.polyu.edu.hk
[2] University of Macau, Av. Padre Tomás Pereira Taipa, Macau
{yb27407,ryanlhu}@umac.mo

Abstract. Many time series algorithms reduce the computation cost by pruning unpromising candidates with lower-bound distance functions. In this paper, we focus on an orthogonal research direction that further boosts the performance by unlocking the potentials of modern commodity CPUs. First, we conduct a performance profiling on existing algorithms to understand where does time go. Second, we design vectorized implementations for lower-bound and distance functions that can enjoy characteristics (e.g., data parallelism, caching, branch prediction) provided by CPU. Third, our vectorized methods are general and applicable to many time series problems such as subsequence search, motif discovery and kNN classification. Our experimental study on real datasets shows that our proposal can achieve up to 6 times of speedup.

1 Introduction

Time series data has various applications in medical diagnosis, speech processing, climate analysis, financial analysis, etc. It has attracted extensive research in the literature [4, 9, 14, 20–22, 25, 30]. We illustrate representative problems in Figure 1: (a) the *subsequence search* problem, which takes a query sequence q and finds its most similar subsequence t_c of a time series t, (b) the *motif discovery* problem, which reports the most similar pair of subsequences in a time series t, and (c) the kNN *classification* problem. These problems typically use the Euclidean Distance (ED) and Dynamic Time Warping (DTW) as the similarity measure.

These problems are computation bound rather than disk I/O bound [22]. Many time series algorithms have been evaluated on commodity CPU [4, 9, 14, 20–22, 25, 30] in single machine. These works focus on devising lower-bound distance functions to prune unpromising candidates and thus reduce calling expensive distance computations.

Even with these effective lower bounds, the above time series problems are still computation intensive, especially for increasingly long time series nowadays (e.g., medical physiological signals[3]). For example, the subsequence search on a trillion scale time series [22] would take 3.1 hours (under the Euclidean distance) and 34 hours (under Dynamic Time Warping) on a commodity PC.

[3] http://www.physionet.org/physiobank/

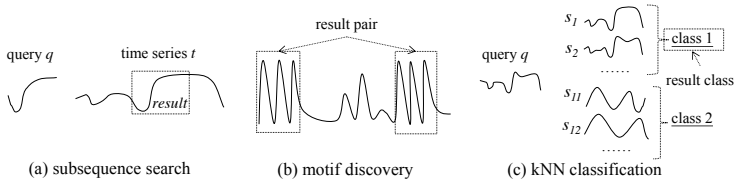


Fig. 1: Problems on time series data

Nevertheless, existing techniques overlook the characteristics of CPU and they have not studied the effect of those characteristics on the CPU time. In general, the CPU time consists of (i) busy cycles, for executing instructions, and (ii) stall cycles, for waiting for instructions or data.

We raise the following questions:

Q1: "In these algorithms, where does time go?"

To answer this question, we profile the performance [3, 27] of existing time-series algorithms (cf. Section 3). Surprisingly, most of the CPU time (70%) is spent on stalling.

Q2: "What cause CPU stall cycles?"

According to our performance profiling, the CPU stall is mainly (more than 80%) caused by branch mispredictions, cache misses, and ALU stall in lower-bound and distance functions.

Q3: "How to reduce CPU stall cycles in modern CPUs?"

Modern CPUs have built-in hardware for branch prediction, caching, and processing vector data efficiently (through SIMD instructions). Recent researches have utilized these characteristics to offer speedup on different problems like join [10], sorting [11], set intersection [16]. In this paper, we will design efficient implementations for lower-bound and distance functions by exploiting the characteristics of modern commodity CPUs. Note that our research direction is orthogonal to the development of lower-bound functions [4, 9, 14, 20–22, 25, 30]. Besides, our proposed techniques are also applicable to mobile time series applications (e.g., continuous heart rate monitoring on Apple watch) as Apple mobile processors (e.g., A5) have supported advanced SIMD instructions since 2011[4].

Our proposed techniques achieve performance gain through: (i) reducing branch mispredictions and cache misses, (ii) incorporating parallelism for vector processing in our computations. We then elaborate these issues in the following two paragraphs.

Conditional branches (e.g., if-then-else, case statements) are commonly used in the lower-bound and distance functions on time series. With branch prediction, a CPU can speculatively execute one path of a conditional branch. A correct prediction can improve the performance due to the CPU's instruction pipeline. However, if the prediction is wrong (i.e., *branch misprediction*), then many CPU cycles will be wasted to flush the instruction pipeline, flush and fetch the relevant data, and restart the execution for the other branch. Therefore, it is desirable to rewrite algorithms to use fewer branching statements and avoid cache pollution. Also, we need to reduce non-compulsory cache misses brought by random memory accesses in our algorithms.

[4] `https://en.wikipedia.org/wiki/Apple_mobile_application_processors`

Data-intensive functions, like lower-bound and distance functions on time series, execute certain arithmetic operations (e.g., multiplication, division) that incur many CPU cycles and thus cause ALU stall. To reduce ALU stall, we use SIMD instructions to process multiple data values per instruction. For example, a SIMD division instruction takes two vectors of values V_a and V_b as input, and perform division $V_a[i]/V_b[i]$ for each position i simultaneously. In this paper, we present vectorized implementations for lower-bound and distance functions by using SIMD. In addition, our vectorized implementations are designed to avoid using conditional branches.

Besides, our proposed techniques are generic and applicable to many time series problems (e.g., subsequence search, motif discovery, kNN classification). In summary, our contributions are:

- We profile the performance of existing time series algorithms and summarize the key insights. ⟨Section 3⟩
- We design vectorized implementations for lower-bound and distance functions. They incur fewer branch mispredictions, cache misses, and ALU stall. ⟨Section 4⟩
- We evaluate the efficiency of our proposed techniques for different time series algorithms on different datasets. Our techniques can achieve up to 6 times of speedup. ⟨Section 5⟩

The rest of this paper is organized as follows. Section 2 clarifies the preliminaries of our research problem. We present the profiling of existing time series algorithms in Section 3. Then, we propose our vectorized implementations in Section 4, perform experimental evaluation on existing time series algorithms in Section 5. Finally, we discuss the related work in Section 6, and conclude this paper in Section 7.

2 Preliminaries

2.1 Fundamental Distance Measurement

In this work, we consider two most popular distance functions, i.e., Euclidean Distance (ED) and Dynamic Time Warping (DTW), in time series problems [13,18,19,21,22,24]. We follow the suggestion from prior literatures [19, 22] that every subsequence should be Z-normalized in order to capture the similarity between the shapes of the sequences. Formally, the i-th value of a Z-normalized sequence $\hat{q}$ can be calculated by $\hat{q}[i] = \frac{q[i]-\mu_q}{\sigma_q}$, where μ_q and σ_q are the mean and standard deviation of q, respectively, and $q[i]$ indicates the i-th element of q. For ease of presentation, we use $dist(q, t)$ to denote the distance $dist(\hat{q}, \hat{t})$ between Z-normalized subsequences in this paper.

Euclidean Distance: This is the most common similarity metric in time series [13, 19, 22, 25, 30] due to its simplicity. We give the definition of squared ED[5] in Equation 1. It takes $O(m)$ time for a query q of length m.

$$ED(q, t_c) = \sum_{i=1}^{m} (\hat{q}[i] - \hat{t}_c[i])^2 \tag{1}$$

[5] The squared distance preserves the relative ordering of distances, and it avoids expensive square root calculations.

Dynamic Time Warping: DTW can capture the similarity of two sequences which may vary in time or have missing values. It is shown to be effective in time series applications [5, 18, 24]. DTW aims to find the optimal alignment (i.e., minimum distance) between two sequences, according to the following recursive equation.

$$DTW(q, t_c) = (\hat{q}[1] - \hat{t}_c[1])^2 + \min \begin{cases} DTW(\hat{q}[2...last], \hat{t}_c) \\ DTW(\hat{q}[2...last], \hat{t}_c[2...last]) \\ DTW(\hat{q}, \hat{t}_c[2...last]) \end{cases} \quad (2)$$

where $\hat{q}[2...last]$ denotes the subsequence of $\hat{q}$ containing values from the 2^{nd} to the last offset. To avoid pathological warping (and reduce the computational cost), the literature [22] suggests to limit the warping length r such that $\hat{q}[i]$ can be matched with $\hat{t}_c[j]$ when $|i - j| \leq r$. This reduces the time complexity of DTW from $O(m^2)$ to $O(mr)$.

2.2 Time Series Algorithms

In Table 1, we summarize the computation techniques (e.g., lower-bounds functions and distance functions) that can be used in three representative time series problems: subsequence search, motif discovery, and classification. Where LB prefixed function provides a lower bound of the exact distance.

Table 1: Computation techniques and distance functions used in time series problems

problem	technique(s)	distance
subsequence search	early distance stop	ED
	$LB_{Kim\mathbf{FL}}, LB^{EQ}_{Keogh}, LB^{EC}_{Keogh}$	DTW
motif discovery	LB_{ref} (uses reference indices)	ED
classification (by kNN)	early distance stop	ED
	$LB_{Kim\mathbf{FL}}, LB^{EQ}_{Keogh}, LB^{EC}_{Keogh}$	DTW

Subsequence search. Formally, given a time series t of length n, a query q of length m, and a distance function $dist(\cdot)$, the subsequence search problem returns a length-m subsequence $t_c \in t$ such that $dist(q, t_c)$ is the minimum (among all length-m subsequences in t).

To the best of our knowledge, UCR Suite [22] is the state-of-the-art solution for the subsequence search problem. It adopts the filter-and-refinement paradigm to reduce exact distance computations. Let bsf be the best-so-far distance obtained during the search process. For ED subsequence search, UCR Suite does not apply any lower-bound function. It accumulates the distance step-by-step and early stops the distance computation $dist(q, t_c)$ as soon as the accumulated value exceeds bsf. For DTW subsequence search, UCR Suite examines each candidate subsequence t_c and applies lower-bound functions on t_c in ascending order of their computation cost: first $LB_{Kim\mathbf{FL}}$, then LB^{EQ}_{Keogh} and finally LB^{EC}_{Keogh}. t_c gets pruned as soon as some $LB(q, t_c)$ exceeds bsf. If t_c survives, then UCR Suite executes the distance function on t_c. We proceed to introduce these lower-bound functions as follows.

$\underline{LB_{Kim\mathbf{FL}}}$ is derived from the **F**irst and the **L**ast sequence values, taking only $O(1)$ time to compute. It is defined as

$$LB_{KimFL}(q, t_c) = (\hat{q}[1] - \hat{t}_c[1])^2 + (\hat{q}[m] - \hat{t}_c[m])^2 \quad (3)$$

$\underline{LB^{EQ}_{Keogh}}$ is derived from the distance between the candidate subsequence $\hat{t}_c$ and the envelop of $\hat{q}$. Given the warping length r, the upper and lower envelop of $\hat{q}$ are defined as $\hat{q}^u[i] = \max_{j=i-r}^{i+r} \hat{q}[j]$ and $\hat{q}^l[i] = \min_{j=i-r}^{i+r} \hat{q}[j]$, respectively, Accordingly, we have

$$LB^{EQ}_{Keogh}(q, t_c) = \sum_{i=1}^{m} \begin{cases} (\hat{t}_c[i] - \hat{q}^u[i])^2 & \text{if } \hat{t}_c[i] > \hat{q}^u[i] \\ (\hat{t}_c[i] - \hat{q}^l[i])^2 & \text{if } \hat{t}_c[i] < \hat{q}^l[i] \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

$\underline{LB^{EC}_{Keogh}}$ is derived similarly to LB^{EQ}_{Keogh} but the lower-bound is derived from the distance between the query and the envelop of $\hat{t}_c$ (i.e., switching roles).

Motif Discovery. Formally, given a time series t of length n, and a query length m, the motif discovery problem returns a pair of length-m subsequences $t_c, t'_c \in t$ such that the Euclidean distance $ED(t_c, t'_c)$ is the minimum among all pairs.

MK [20] is a representative solution for motif discovery. To avoid examining every subsequence pair, it proposes a reference based lower-bound. Given a set of subsequences and their distances to a set of references R, the lower-bound of two subsequences t_a and t_b can be derived as follows.

$$LB_{ref}(t_a, t_b) = \max_{r_i \in R} |distRef[r_i][t_a] - distRef[r_i][t_b]| \quad (5)$$

where $distRef[r_i][t] = ED(r_i, t)$.

MK first constructs a sorted list of every subsequence in terms of their distances to a reference. Intuitively, if the lower-bound of every 1^{st} neighbor pair (in terms of their positions in the sorted list) is worse than bsf, then it is not necessary to examine further neighbor pairs (e.g., 2^{nd} neighbor pairs) due to the monotonicity of the sorted list. Thereby, MK iteratively examines the subsequence pairs based on their sorted list positions. At the end of an iteration, the search terminates when no neighbor pair has lower-bound better than bsf.

Classification. ED and DTW are widely accepted for describing the similarity between time series in the classification problem [12]. We can apply the same techniques for subsequence search (i.e., early distance stop for ED and lower-bound techniques for DTW) to boost the classification process.

2.3 Modern Commodity CPUs

Modern commodity CPUs share the following hardware characteristics that can be further exploited in algorithm design.

- (1) **Single instruction multiple data**: Modern commodity CPUs provide vector instructions (SIMD) operating on 256-bit vector registers that allow to perform the same instruction on multiple data values in parallel.
- (2) **Hardware prefetcher**: Modern commodity CPUs have built-in hardware prefetcher. It allows to prefetch additional lines of instruction or data into the L1 or L2 cache in CPU cores.

The modern commodity CPUs also have multiple cores and simultaneous multi-threading technique. We leave the study on multi-threading issues for time series algorithms as future work. All algorithms in this paper run in single thread model by default.

3 Profiling of Algorithms

We first describe our experimental platform and then present the profiling result on existing time series algorithms.

3.1 Experimental Setting

In all experiments, we use a machine with a 3.40GHz Intel(R) Core(TM) i7-4770 CPU based on Haswell micro-architecture, 16 GB main memory, and a SSD (solid state drive, 256GB capacity, 545 MB/s sequential read throughput). The CPU has 4 physical cores and supports simultaneous multithreading. The machine runs Ubuntu 14.04. All algorithms have been implemented in C++ and compiled by GNU C++ compiler with level 3 optimization.

We use the following real datasets and list their information in Table 2. All datasets are stored in the SSD.

- For the subsequence search problem, we use three datasets. Both **ECG-E**[6] and **ECG-L**[7] are electrocardiography (ECG) recordings, and we use the same query sequences (of length 421) as in [22] as the default query sequences. **EEG-C**[8] contains electroencephalography (EEG) recordings, and we randomly extract query sequences (of length 128) from the epileptic seizure recording as in [26]. For each dataset, we follow the experimental methodology in [22], and obtain a single time series by concatenating all data sequences.
- For the kNN classification problem, we use **Weather** [9] dataset, which contains the temperature data extracted from weather forecast records. It contains 11,508 sequences, each sequence in Weather corresponds to a one-year time series collected from 5,936 locations. We use the attribute "Country" as the class attribute. We randomly choose data sequences as queries and exclude them from the data.
- For the motif discovery problem, we use two datasets: **EEG-MK**[10] and **TAO-MK** [19].

3.2 Measurement Methodology

Program execution time: According to the Intel performance analysis manual [1], the program execution time (T_R) consists of: computation time (T_C), branch misprediction stall (T_{Br}), backend stall (T_{Be}), and frontend stall (T_{Fe}). The computation time (T_C) is regarded as '*CPU busy*', and the rest as '*CPU stall*'. The backend stall occurs when the requested resource is being held-up in back end. It includes ALU stall (T_{ALU}) and memory stall (T_{Cache}). T_{ALU} is the ALU execution unit stall, which is caused by the execution of arithmetic operations (e.g., divide, square root) that require many cycles. T_{Cache} is the memory-bound stall, which is caused by L1 data cache misses, L2 cache misses, L3 cache misses or TLB cache misses.

[6] http://www.physionet.org/physiobank/database/edb/

[7] http://www.physionet.org/physiobank/database/ltstdb/

[8] http://www.physionet.org/pn6/chbmit/

[9] http://data.gov.uk/metoffice-data-archive

[10] http://www.cs.ucr.edu/∼mueen/OnlineMotif/index.html

Table 2: Dataset information

dataset	sequence length	data size	problem
ECG-E	$1.60 \cdot 10^8$	611 MB	subsequence search
ECG-L	$1.89 \cdot 10^9$	7.06 GB	
EEG-C	$1.01 \cdot 10^{10}$	37.5 GB	
EEG-MK	$1.80 \cdot 10^5$	704 KB	motif discovery
TAO-MK	$7.42 \cdot 10^5$	2.82 MB	
Weather	$1.81 \cdot 10^3$,	19.86 MB	kNN classification

We summarize the breakdown of execution time in a CPU as follows:

$$T_R = T_C + T_{stall}; \text{ where } T_{Stall} = T_{Br} + T_{ALU} + T_{Cache} + T_{Fe}$$

Profiling experiments: To measure the above components of CPU time, we used PAPI [8] to obtain hardware performance counters from CPU, e.g., the number of stall cycles and the number of CPU cycles. In each subsequence search and classification experiment, we report the average CPU time over 10 queries. To ensure the confidence level, we repeat running each query until the maximum standard deviation of the important counters (UOPS_RETIRED:RETIRE_SLOTS, CPU_CLK_UNHALTED:THREAD_P) is less than 3%.

Experimental reproducibility: For the sake of experimental reproducibility, we have posted the datasets and source codes at [2] [11].

3.3 Identifying the Performance Bottleneck

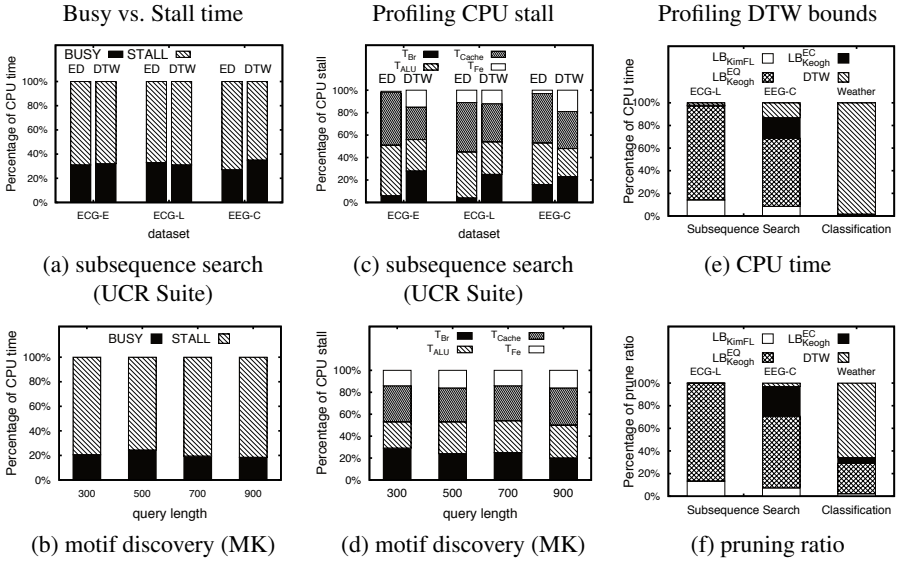
In this section, we profile the performance of existing solutions and then identify the performance bottleneck. We conduct experiments to profile the performance of representative solutions: (i) UCR Suite [22] for the subsequence search problem, (ii) MK [20] for the motif discovery problem, and (iii) kNN classification [22] for the classification problem.

CPU stall & CPU busy: Figures 2(a) and (b) report the CPU time breakdown of existing solutions into *busy* time and *stall* time, for subsequence search and motif discovery, respectively.
Observation: The majority (65–70%) of the CPU time is spent on stalling (i.e., wasted CPU cycles).

CPU stall breakdown: We then delve into CPU stall and plot the breakdown of CPU stall time in Figures 2(c) and (d).
Observation: The CPU stall is dominated (more than 80%) by ALU stall, cache misses, and branch mispredictions penalties.

CPU time of different functions: The DTW function and its lower-bound functions ($LB_{Kim\mathbf{FL}}, LB^{EQ}_{Keogh}, LB^{EC}_{Keogh}$) are applicable to the subsequence search problem and the classification problem [22]. We profile the performance of [22] on these two problems in Figure 2(e). Different functions incur different portions of time and pruning ratio (cf. Figure 2(f)) in different scenarios. For example, lower-bound functions $LB^{EQ}_{Keogh}, LB^{EC}_{Keogh}$ dominate the time for subsequence search. However, the DTW computation incurs more time in kNN classification problems.

[11] For consistency, we use the 'float' data type to represent time series values in all evaluated methods.



(a) subsequence search (UCR Suite)
(c) subsequence search (UCR Suite)
(e) CPU time
(b) motif discovery (MK)
(d) motif discovery (MK)
(f) pruning ratio

Fig. 2: Profile existing solutions

Observation: Different time series problems spend very different proportions of time on different functions. Therefore, it is important to optimize the computation of both lower-bound functions LB^{EQ}_{Keogh}, LB^{EC}_{Keogh} and the DTW function.

4 Accelerating Distance Functions with SIMD

As shown in the previous section, the majority of CPU stall is caused by ALU stall, cache misses and branch mispredictions. In this section, we will design vectorized implementations for exact distance and lower-bounds functions to reduce those stalls. We will also evaluate the efficiency of our implementations with experiments.

4.1 How do SIMD instructions reduce stall?

SIMD vectorization: reduce ALU stall The ALU stall is caused by the execution of arithmetic operations that require many CPU cycles. For example, the 'division' instruction for two floating-point values takes 24 CPU cycles [1].

Modern CPU provides SIMD instructions to perform the same instruction (e.g, $+, -, \times, /, \min, \max$) on multiple data values in parallel. For instance, Intel i7-4770 and AMD Phenom II support the AVX2 instruction set (SIMD instructions on 256-bit registers). The SIMD instruction `simd_div` (e.g., `_mm256_div_ps` in AVX2) performs division on 8 pairs of values in two SIMD registers R_a and R_b simultaneously. It takes only 21 CPU cycles [1], which is much cheaper than executing the 'division' instruction on 8 pairs one-by-one (using 24*8=192 cycles). Thus, SIMD instructions help reduce the ALU stall significantly.

Distance computation indeed fits well with SIMD instructions. As we illustrate in Figure 3, we may divide subsequences into groups of length 8, and then apply SIMD instructions on each group to compute distances for pairs.

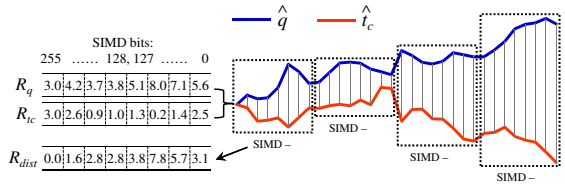
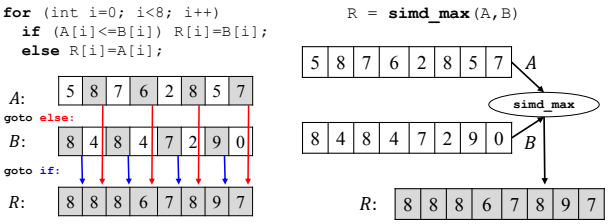


Fig. 3: Using SIMD for distance computation

Typical SIMD width: Our CPU (Intel i7-4770) is a modern commodity CPU. It supports the following SIMD widths and instruction sets: (i) 64 bits (i.e., MMX instruction set), (ii) 128 bits (i.e., SSE instruction set), (iii) 256 bits (i.e., AVX instruction set). Since the MMX instruction set does not support floating-point values, it cannot be used in time series problems. Thus, we report the results for 128 bits (SIMD-128) and 256 bits (SIMD-256) in following experiments. For simplicity, we set 256 bits (SIMD-256) as default SIMD register.

Hardware prefetching: reduce branch misprediction Modern CPU is equipped with a branch prediction unit and it speculatively executes a conditional branch to maximize the utilization of CPU resources. A correct prediction can improve the performance due to the built-in instruction pipeline and hardware prefetching. However, incorrect prediction will bring cache pollution[12] and waste CPU cycles to flush instructions and restart execution.



(a) code fragment and schematic for if-else statement

(b) code fragment and schematic for SIMD max instruction

Fig. 4: Example for reducing branching statements

Some SIMD instructions help reduce branch misprediction. For example, for the code fragment in Figure 4(a), the CPU may incur up to 8 branch mispredictions in the worst case. In contrast, the alternative implementation in Figure 4(b) has no branch mispredictions because it uses a single instruction `simd_max` instead of conditional branches.

We observe that DTW and its lower-bound functions (cf. Section 2) have many conditional branches. Therefore, we need to design SIMD implementations for DTW and its lower-bound functions without using conditional branches.

[12] http://en.wikipedia.org/wiki/Cache_pollution

4.2 Accelerating ED with SIMD

Before presenting our SIMD solutions, we first introduce the existing implementation of Euclidean distance. We call it as SISD-ED (cf. Algorithm 1) because it uses traditional CPU instructions, i.e., *Single Instruction, Single Data* (SISD). According to Section 2, we perform Z-normalization on the subsequence t_c (cf. Line 3). It early stops the computation if the accumulated distance $dist$ exceeds the best-so-far distance bsf (cf. Line 5).

Algorithm 1 SISD-$ED(q, t_c)$

Input: best-so-far bsf, mean μ and stdev. σ of candidate t_c,
Output: squared distance $dist$
1: $dist := 0$
2: **for** $i := 1$ to m **do**
3: $\quad c := (t_c[i] - \mu)/\sigma$ ▷ Z-normalization
4: $\quad dist := dist + (c - \hat{q}[idx])^2$ ▷ accumulation
5: $\quad$ **if** $dist \geq bsf$ **break** ▷ early stop
6: return $dist$

Next we demonstrate how we employ SIMD to accelerate $ED(\cdot)$ in different steps. The intuition is to compute 8 offsets between q and t_c by batch. In the Z-normalization step, we can normalize 8 offset values simultaneously as follows.

SIMD Z-normalization
1: $R_c :=$ `simd_load`$(\&t_c[i])$ ▷ load t_c
2: $R_c :=$ `simd_sub`(R_c, R_μ) ▷ vectorized $t_c[i] - \mu$
3: $R_c :=$ `simd_div`(R_c, R_σ) ▷ vectorized $(t_c[i] - \mu)/\sigma$

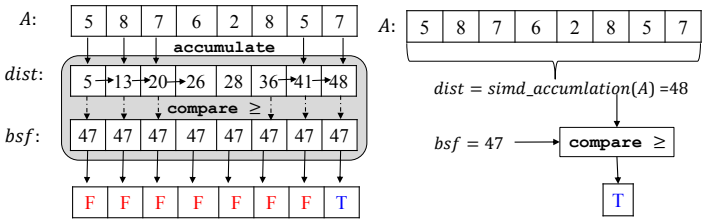
where R_c, R_μ, R_σ are the corresponding SIMD registers of variables c, μ, and σ, respectively. Note that each register stores 8 floating-point values. In the accumulation step, we can compute the distance of 8 offsets $(\hat{t}[i] - \hat{q}[i])^2$ as follows.

SIMD distance computation
1: $R_{\hat{q}} :=$ `simd_load`$(\&\hat{q}[i])$ ▷ load $\hat{q}[0]$...$\hat{q}[7]$
2: $R_d :=$ `simd_sub`$(R_{\hat{q}}, R_c)$ ▷ vectorized $\hat{t}[i] - \hat{q}[i]$
3: $R_d :=$ `simd_mul`(R_d, R_d) ▷ vectorized $(\hat{t}[i] - \hat{q}[i])^2$

Before examining the early stop condition (cf. Line 5 of Algorithm 1), we need to accumulate 8 offset distances into $dist$. Since the AVX2 instruction set has no single instruction to accumulate the values of an SIMD register, we accomplish the accumulation by the following sequence of SIMD instructions.

SIMD distance accumulation
1: $R_d :=$ `simd_hadd`(R_d, R_d) ▷ add horizontal pairs
2: $R_d :=$ `simd_hadd`(R_d, R_d) ▷ add horizontal pairs
3: $S_d :=$ `simd_extractf`$(R_d, 1)$
4: $S_d :=$ `simd_sadd`$($`simd_cast`$(R_d), S_d)$
5: $dist := dist +$ `simd_scvt`(S_d)

The accumulation employs instruction `simd_hadd` (e.g., `_mm256_hadd_ps`) twice that horizontally adds adjacent pairs of 32-bit floating-point elements in the input registers, and stores the results into an output register. Then decompose the vector into two parts by `simd_extractf` and `simd_cast`. Next, we sum the first value of



(a) SISD-ED (b) SIMD-ED

Fig. 5: Example for early stop

two decomposed vectors (by `simd_sadd`), extract the lower 32-bit floating-point element from the vector (by `simd_scvt`), and accumulate it into $dist$. The accumulation process takes logarithmic cost to the SIMD register length.

Our vectorized implementation reduces CPU cycles by (i) incorporating parallelism for Z-normalization and distance computation, and (ii) reducing branching statements for the early stop condition. Figure 5(a) shows that SISD-ED requires verifying the early stop for every accumulation (i.e., 8 comparisons in total). In SIMD-ED, we only verify the early termination once per 8 accumulations as shown in Figure 5(b).

4.3 Accelerating DTW with SIMD

For the sake of our discussion, we first present the pseudo code of DTW computation in Algorithm 2. It employs a matrix $C[1..m][1..m]$ whose entry $C[i][j]$ is used to store the DTW value between subsequences $\hat{q}[1..i]$ and $\hat{t_c}[1..j]$. Then, we fill the matrix C by row-by-row ordering (Lines 2–3). Observe that we cannot compute values in the same row (e.g., $C[i][j-1], C[i][j]$) in parallel because $C[i][j]$ depends on $C[i][j-1]$.

Algorithm 2 SISD-$DTW(q, t_c)$

Input: warping constraint length r, normalized query $\hat{q}$ and candidate $\hat{t}_c$
Output: squared distance $dist$
1: Distance array $C[1..m][1..m]$, initialized to $+\infty$
2: **for** $i := 1$ to m **do**
3: **for** $j := \max(0, i-r)$ to $\min(m, i+r)$ **do**
4: **if** $i = 1$ and $j = 1$ **then**
5: $C[1][1] := (\hat{q}[1] - \hat{t}_c[1])^2$
6: **else**
7: $C[i][j] := (\hat{q}[i] - \hat{t}_c[j])^2 + \min(C[i-1][j], C[i-1][j-1], C[i][j-1])$
8: return $C[m][m]$ as $dist$

To better utilize SIMD instructions, we rewrite the equation of $C[i][j]$ into an alternative form as follows.

$$C[i][j] = (\hat{q}[i] - \hat{t}_c[j])^2 + \min(B_{i-1}[j], C[i][j-1]) \tag{6}$$

where $B_{i-1}[j] = \min(C[i-1][j-1], C[i-1][j])$. Since $B_{i-1}[j]$ depends only on values in the previous row of C (i.e., row $i-1$), we can calculate consecutive values of B_{i-1} (e.g., $B_{i-1}[j]$ to $B_{i-1}[j+7]$) in a batch.

The above discussion enables us to rewrite Line 7 in SISD-DTW as the following pseudo code using SIMD instructions.

Rewrite inner for-loop (i fixed) in Algorithm 2

1: $j_{min} := \max(0, i-r); j_{max} := \min(m, i+r)$
2: **for** $j := j_{min}$ to j_{max}, 8 offsets **do**
3: load R_{x1} with $C[i-1][j, \cdots, j+7]$
4: load R_{x2} with $C[i-1][j-1, \cdots, j+6]$
5: $R_B := \texttt{simd\_min}(R_{x1}, R_{x2})$
6: $B_{i-1}[j, \cdots, j+7] = \texttt{simd\_store}(R_B)$
7: $C[i][j] := (\hat{q}[i] - \hat{t}_c[j])^2$
8: $C[i][j_{min}] := C[i][j_{min}] + B_{i-1}[j_{min}]$
9: **for** $j := j_{min} + 1$ to j_{max} **do**
10: increment $C[i][j]$ by $\min(C[i][j-1], B_{i-1}[j])$

The rewritten code has two nice properties: (i) avoid branch mispredictions by using the `simd_min` instruction (Lines 3–5), (ii) reduce cache misses by utilizing the data locality of $C[i][j-1]$ and $C[i][j]$ (Line 10). Figure 6 illustrates how our SIMD implementation works (when $i = 4$).

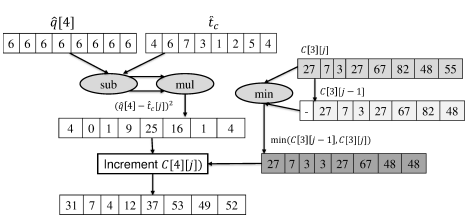


Fig. 6: SIMD DTW illustration, at $i = 4$

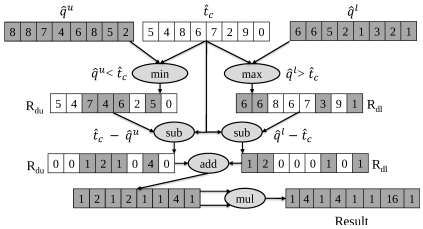


Fig. 7: LB^{EQ}_{Keogh} SIMD illustration

Optimized implementation: For ease of understanding, we employ $m \times m$ matrices in the above algorithms. An optimized implementation is to use 2 float arrays with size $2r + 1$ (i.e., store $C[i-1]$ and $C[i]$ in Line 7, Algorithm 2) to compute DTW (for both SISD-DTW and SIMD-DTW). Since these 2 float arrays can fit in low latency cache (e.g., L2 cache rather than L3 cache), we use this optimized implementation for both SISD-DTW and SIMD-DTW in our code.

4.4 Accelerating Lower Bounds for DTW with SIMD

We proceed to present SIMD optimizations for lower-bound functions LB^{EQ}_{Keogh} and LB^{EC}_{Keogh}. Since these two functions are similar, our discussion focuses on LB^{EQ}_{Keogh}.

Similar to $ED(\cdot)$, we present the SISD implementation of LB^{EQ}_{Keogh} in Algorithm 3. It derives the lower-bound lb from the candidate subsequence t_c and the envelop of q which is handled by the **if-then-else** statement at Lines 4-7. However, the **if-then-else** statement may cause many branch mispredictions in CPU, leading to high stalling time (e.g., 10–20 clock cycles in modern CPU on average). In addition, as reported in [15], the hardware prefetching (for reducing cache misses) technique becomes less effective in the presence of multiple code paths.

To avoid branch mispredictions and better utilize hardware prefetching, we should remove branching, i.e., the **if-then-else** statement, in Algorithm 3. Lemma 1 shows the alternative form of LB^{EQ}_{Keogh} (cf. Equation 4 in Section 2).

Algorithm 3 SISD-$LB^{EQ}_{Keogh}(q, t_c)$

Input: best-so-far bsf, mean μ and stdev. σ of candidate t_c, upper and lower envelops $\hat{q}^u$ and $\hat{q}^l$
Output: lower-bound distance lb

$lb := 0$
for $i := 1$ to m **do**
  $c := (t_c[i] - \mu)/\sigma$ ▷ Z-normalization
  if $\hat{q}^u[i] < c$ **then** ▷ distance of $\hat{t}_c$ and the envelop of $\hat{q}$
    $lb := lb + (c - \hat{q}^u[i])^2$
  else if $\hat{q}^l[i] > c$ **then**
    $lb := lb + (\hat{q}^l[i] - c)^2$
  if $dist \geq bsf$ **break** ▷ early stop
return lb

Lemma 1 (Alternative form of LB^{EQ}_{Keogh}).

$$LB^{EQ}_{Keogh} = \sum_{i=1}^{m}((\hat{t_c}[i] - \min\{\hat{t}_c[i], \hat{q}^u[i]\}) + (\max\{\hat{t}_c[i], \hat{q}^l[i]\} - \hat{t}_c[i]))^2$$

Proof. LB^{EQ}_{Keogh} (cf. Equation 4) consists of three cases.

Case 1: When $\hat{t}_c[i] < \hat{q}^u[i]$, the first part (i.e., $\hat{t_c}[i] - \min\{\hat{t_c}[i], \hat{q}^u[i]\}$) becomes zero so that the equation reduces to $(\hat{q}^l[i] - \hat{t}_c[i])^2$.

Case 2: When $\hat{t}_c[i] > \hat{q}^l[i]$, the second part (i.e., $\max\{\hat{t}_c[i], \hat{q}^l[i]\} - \hat{t}_c[i]$) becomes zero so that the equation reduces to $(\hat{t}_c[i] - \hat{q}^u[i])^2$.

Case 3: Otherwise, none of the first or the second part contributes so the equation returns 0. □

Since this form uses only $\min, \max, +, -, \times$, we can readily implement them by the corresponding SIMD instructions. Accordingly, the first part and the second part of LB^{EQ}_{Keogh} can be computed as follows. Then, we sum up both parts.

SIMD $\hat{t_c}[i] - \min\{\hat{t}_c[i], \hat{q}^u[i]\}$ computation, 8 offsets

$R_{\hat{q}^u}$:= `simd_load`(&$\hat{q}^u[i]$) ▷ vectorized load $\hat{q}^u[i]..\hat{q}^u[i+7]$
R_{du} := `simd_min`($R_{\hat{q}^u}, R_c$) ▷ vectorized $\min\{\hat{t_c}[i], \hat{q}^u[i]\}$
R_{du} := `simd_sub`($R_c, R_{\hat{q}^u}$) ▷ vectorized $\hat{t_c}[i] - \min\{\hat{t_c}[i], \hat{q}^u[i]\}$

SIMD $\max\{\hat{t}_c[i], \hat{q}^l[i]\} - \hat{t}_c[i]$ computation, 8 offsets

$R_{\hat{q}^l}$:= `simd_load`(&$\hat{q}^l[i]$) ▷ vectorized load $\hat{q}^l[i]..\hat{q}^l[i+7]$
R_{dl} := `simd_max`($R_{\hat{q}^l}, R_c$) ▷ vectorized $\max\{\hat{t}_c[i], \hat{q}^l[i]\}$
R_{dl} := `simd_sub`($R_{\hat{q}^l}, R_c$) ▷ vectorized $\max\{\hat{t}_c[i], \hat{q}^l[i]\} - \hat{t}_c[i]$

SIMD combining the result of R_{du} and R_{dl}, 8 offsets

R_d := `simd_add`(R_{du}, R_{dl}) ▷ vectorized sum
R_d := `simd_mul`(R_d, R_d) ▷ vectorized square

We illustrate our idea by a concrete example in Figure 7. First we extract the *min* values between $\hat{q}^u$ and $\hat{t}_c$ of 8 offsets by `simd_min` and then store them into R_{du}. Next we subtract R_{du} from t_c to finish the first part computation. The second part is performed similarly where the *max* values are stored into R_{dl}. Next we combine the distance values from R_{du} and R_{dl} to produce R_d. Finally we multiply the values of R_d to generate the squared distance, and then execute SIMD distance accumulation as described in Section 4.2.

4.5 Cost analysis

We proceed to analyze the cost of the SISD and SIMD implementations based on the latency cycle information given in the Intel architecture optimization manual [1].

ED: Our analysis covers four steps in ED: (i) Z-normalization, (ii) distance computation, (iii) distance accumulation, and (iv) early stop, as shown in Table 3(a). In each step, we list all used instructions and their latency cycles. For SIMD-ED, the denominator in latency is 8 as it processes 8 offset values simultaneously. In summary, SIMD-ED is $41/7 = 5.86$ times faster than SISD-ED.

DTW: We analyze the latency of both SISD and SIMD implementations of DTW in Table 3(b). The speedup of the SIMD implementation over SISD one is: $\frac{48}{14.625} = 3.28$.

LB^{EQ}_{Keogh}: We analyze the latency for two implementations of LB^{EQ}_{Keogh}. We only list the detail cost at step (ii) distance computation in Table 3(c) as the other three steps are the same as in SIMD-ED (cf. Table 3(a)). SIMD-LB^{EQ}_{Keogh} outperforms SISD-LB^{EQ}_{Keogh} by $43/9 = 4.78$ times.

Table 3: Instruction latency of SISD and SIMD functions

Step		SISD-	SIMD-
Z-norm.	op cost	`load`, −, / 1+3+24 = 28	`gat.`, `sub`, `div` (1+3+21)/8 = 25/8
Dist. compu.	op cost	`load`, −, × 1+3+5 = 9	`load`, `sub`, `mul` (4+3+5)/8 = 12/8
Accum.	op	+	2·`hadd`, 2·`add`, `ext.`, `scvt`, `cast`
	cost	3	(2*5+2*3+1*2)/8 = 18/8
Ear. stop	cost	1	1/8
Total	cost	41	7

(a) ED

Step		SISD-	SIMD-
ED (cf. (a))	cost	28+9+3 = 40	(25+12+18)/8 =55/8
Take Mini.	op	3*`load`, 2*`cmp`	2*`load`, `min`,`store`
	cost	3*1+2= 5	(2*4+3+3)/8 = 14/8
Accum.	op	+	2*`load`, 1*`cmp`, +
	cost	3	2*1+1+3 = 6
Total		48	14.625

(b) DTW

Step		SISD-	SIMD-
Dist. compu.	op	`load` 2·`cmp`, −, ×	2·`load`, 2·`sub`, `min`, `mul`, `max`, `add`
	cost	1+2+3+5 =11	(8+6+3+3+3+5) /8=28/8
Z-norm. Accum. Ear. stop (cf. (a))	cost	28+3+1	(25+18+1)/8
Total		43	9

(c) LB^{EQ}_{Keogh}

4.6 Accelerating Reference Index with SIMD

Before proposing our SIMD solution, we first present the existing implementation of LB_{ref} in Algorithm 4.

Algorithm 4 SISD-$LB_{ref}(t_a, t_b)$

Input: best-so-far bsf, reference distance $dist_{ref}$, # of reference R, two subsequences t_a, t_b
Output: Boolean value
1: **for** $i := 1$ to R **do**
2: **if** $|dist_{ref}[i][a] - dist_{ref}[i][b]| > bsf$ **then**
3: return true ▷ can be pruned
4: return false ▷ cannot be pruned

As the absolute value computation is not supported by the AVX2 instruction set, we rewrite $|dist_{ref}[i][a] - dist_{ref}[i][b]|$ as:

$$\max(dist_{ref}[i][a], dist_{ref}[i][b]) - \min(dist_{ref}[i][a], dist_{ref}[i][b])$$

Then, we design the SIMD implementation below for LB_{ref}. It avoids using branching statements for early termination in Lines 7-8. We verify the early stop only once by executing `simd_cmp` for 8 pairs of candidates.

SIMD LB_{ref}, for 8 offsets

1: R_a := simd_load($dist_{ref}[i][a], \cdots, dist_{ref}[i+7][a]$)
2: R_b := simd_load($dist_{ref}[i][b], \cdots, dist_{ref}[i+7][b]$)
3: R_{bsf} := simd_set1(bsf)
4: R_{max} := simd_max(R_a, R_b)
5: R_{min} := simd_min(R_a, R_b)
6: R_{sub} := simd_sub(R_{max}, R_{min})
7: R_a := simd_cmp($R_{sub}, R_{bsf}, >$)
8: return simd_testz(R_a, R_a)

We can further optimize LB_{ref} by sequentializing the memory access (and reducing CPU cache misses). This requires changing the memory layout of $dist_{ref}$ to $dist_{ref}[a][i]$ (i.e., swapping the role of rows and columns) so that Lines 1–2 have sequential main memory accesses.

Cost analysis: For each reference point i, SISD-LB_{ref} takes 6 cycles and SIMD-LB_{ref} takes 27/8 cycles. We omit the detailed analysis here.

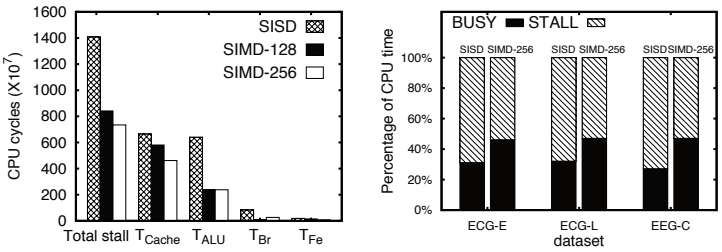
Alternative implementation: Another implementation for $|dist_{ref}[i][a] - dist_{ref}[i][b]|$ is to use simd_sub, simd_set and simd_andnot instructions only. Since this implementation spends the same number of CPU cycles as in the above algorithm, we omit its detail discussion in following experiments.

5 Experimental Study

In this section, we conduct extensive experiments to evaluate our proposed techniques with existing solutions. Unless otherwise stated, we use the experimental platform and measurement methodology in Section 3. Note that the execution time includes both disk I/O time and CPU computation time. We denote SISD as the original implementation [20, 22] (for corresponding problems), SIMD as the implementation with our proposed techniques.

5.1 Subsequence Search

UCR-ED: UCR-ED [22] is a representative solution for the ED-based subsequence search. It employs the *early abandoning* technique to accelerate the Euclidean distance computation. We show the performance of SISD-based and SIMD-based UCR-ED in Figure 8.



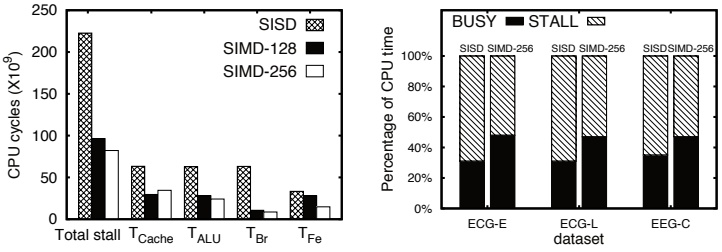
(a) stall breakdown on ECG-E (b) CPU time breakdown

Fig. 8: SISD-based and SIMD-based UCR-ED

First, we investigate the components of CPU stall of the methods on the dataset ECG-E in Figure 8(a). Since our SIMD-based solutions exploit SIMD vectorization techniques, they incur fewer instructions and ALU stall (T_{ALU}) than the SISD-based solution. The results on other datasets are similar to Figure 8(a), so we omit them for space reasons.

Second, we compare the CPU time of the methods in Figure 8(b). We omit the results of SIMD-128, as it is similar to SIMD-256. Clearly, our SIMD-based UCR-ED can reduce CPU stalls significantly (e.g., $\sim$20%).

UCR-DTW: Regarding the DTW-based subsequence search, UCR-DTW [22] cascades three lower bound techniques (i.e., LB_{KimFL}, LB^{EQ}_{Keogh}, and LB^{EC}_{Keogh}) to pruning unpromising candidates without invoking expensive DTW computations. We breakdown the components of CPU stall of the methods on the dataset ECG-E in Figure 9(a). Since our SIMD-based UCR-DTW accelerated both the exact distance (cf. Section 4.3) and the lower bound computations (cf. Section 4.4), SIMD-based UCR-DTW introduces fewer CPU stall cycles than the SISD-based solution. Second, we compared the CPU time of the methods on three datasets in Figure 9(b). The CPU busy ratio of SIMD-based UCR-DTW is almost 50%, which is much higher than SISD-based UCR-DTW.



(a) stall breakdown on ECG-E (b) CPU time breakdown

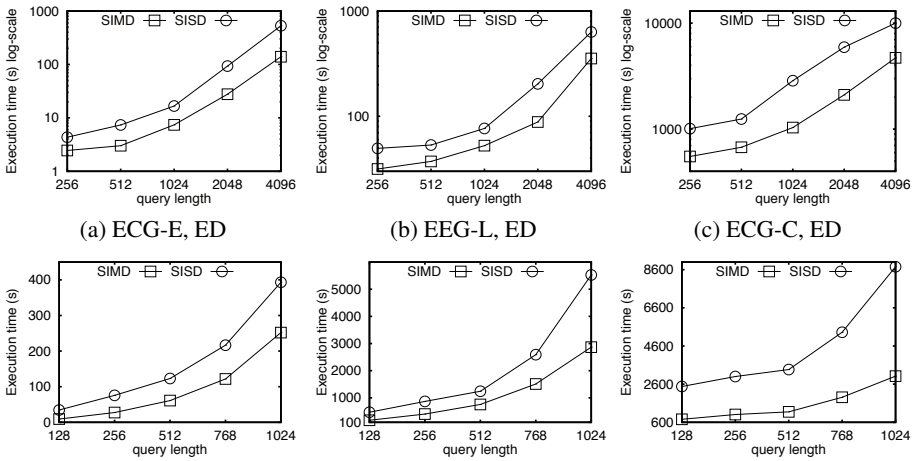
Fig. 9: SISD-based and SIMD-based UCR-DTW

Execution Time Speedup: In this set of experiments, we report the execution time of the methods on three time series datasets (i.e., ECG-E, ECG-L, and EEG-C) varying on query lengths in Figure 10, where the lengths are from 256 to 4096 in UCR-ED and 128 to 1024 in UCR-DTW. Our proposed SIMD-based methods are 1.8-3.8 and 1.5-3.2 times faster than UCR-ED and UCR-DTW, respectively.

5.2 Motif Discovery

MK [20] makes use of (i) the exact distance calculation ED and (ii) the lower-bound calculation LB_{ref}. The SIMD-based implementation of these two functions has been introduced in Section 4.2 and 4.6, respectively. Figure 11(a) illustrates the reduced cycles of each CPU stall component in SISD-based and SIMD-based MK. Figure 11(b) shows the improvement of the CPU cycles with respect to different query lengths. Again, the SIMD-based solution introduces fewer stall cycles as compared with the SISD-based solution.

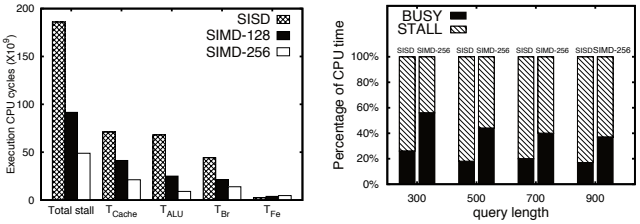
We then compare the performance of the methods on the motif discovery problem. Figure 12 plots the execution time (logscale) of the methods with respect to the query



(a) ECG-E, ED (b) EEG-L, ED (c) ECG-C, ED

(d) ECG-E, DTW (e) ECG-L, DTW (f) EEG-C, DTW

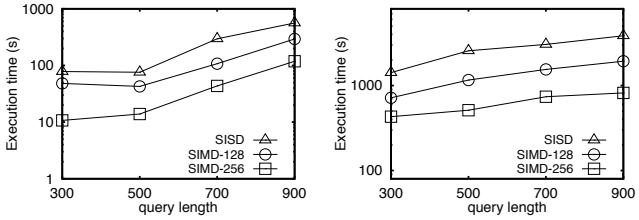
Fig. 10: [Subsequence search] vary query length



(a) stall breakdown on EEG-MK (b) CPU time breakdown

Fig. 11: SISD-based and SIMD-based MK, EEG-MK

length. The performance gap between our methods and SISD widens as the query length increases. The speedup of SIMD over SISD ranges from 2.2 to 6.0.



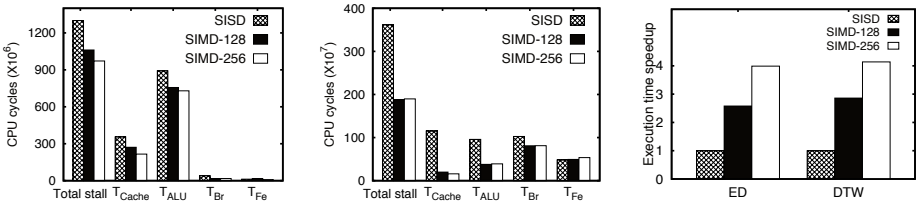
(a) EEG-MK (b) TAO-MK

Fig. 12: [Motif discovery] vary query length

5.3 *k*NN Classification

We show the breakdown of CPU stalls of UCR Suite based kNN classification problem on Weather dataset in Figure 13. We set k=1 which is the default setting in [12]. This

problem is less computational intensive (one candidate per sequence) when compared to the subsequence search problem ($O(n)$ subsequences per sequence) and the motif problem ($O(n^2)$ subsequence pairs per sequence). Even though it is less computational intensive, the SIMD-based solution still saves $\sim$50% stall cycles for DTW as compared to the SISD-based solution (cf. Figure 13). Next we show the execution time speedup of the methods on the kNN classification problem in Figure 13(c). kNN classification problem is less computational intensive, Thus, the speedup by SIMD is lower than before. Nevertheless, SIMD still outperforms all other methods.



(a) UCR-ED based breakdown (b) UCR-DTW based breakdown (c) speedup

Fig. 13: Breakdown of CPU stalls and speedup, kNN Classification

6 Related work

Time series: The representative problems on time series data are (i) the *subsequence search* problem [9, 12, 14, 21, 22, 25, 30], (ii) the *motif discovery* problem [20] and (iii) the kNN *classification* problem [12, 22]. The typical distance functions are the Euclidean distance (ED) and Dynamic Time Warping (DTW). Existing algorithms rely on software-level optimizations such as lower-bound functions and indexing structures [4, 13, 21, 22, 30]. However, existing solutions incur high CPU stall times and there are rooms to further improve the efficiency of distance and lower-bound computations. To our best knowledge, our work is the first to exploit data parallelism to speedup the above computations on modern CPU. Our proposed techniques are orthogonal to the above software-level optimizations.

Modern CPU: Modern CPU provides data parallelism via single instruction over multiple data (SIMD) and offer thread parallelism through multiple cores and simultaneous multi-threading (SMT). In the relational database area, SIMD and multi-core CPUs have been used to speedup database operations [29], sorting [11], and joining [6, 7, 23]. In contrast, we focus on accelerating lower-bound and distance computations on time series data.

Other computing devices: We are aware of methods that accelerate DTW subsequence search on GPUs and FPGAs [24, 28]. They aim at parallelizing the computation of DTW, which however is not always the dominant cost as shown in our performance profiling. Note that the performance of GPU degrades if it works on a dataset much larger than its video RAM (2 GB). The typical bandwidth between GPU and the main memory is 15GB/s, which is much smaller than the bandwidth between CPU and the main memory.

7 Conclusion and Future Work

Summary and Lessons Learnt: In this paper, we conduct performance profiling on existing solutions for time series problems. We find that the performance bottleneck is caused by CPU stalls. We have redesigned vectorized lower-bound and distance functions with SIMD instructions for time series problems. Through our experimental results and analysis, we have two key findings, which will shed light on the design and implementation of time series algorithms on modern commodity CPUs.

Firstly, the performance bottlenecks of different time series applications are different. Even for the same time series algorithm, it may incur different bottlenecks on different datasets, depending on the pruning power of each specific lower bound function. Secondly, the characteristics of modern CPUs (e.g., branch prediction unit, hardware prefetching, vectorization) play important roles in the execution time of an implementation. Frequently-used functions (e.g., lower-bound and exact distance computations) need to be redesigned in order to unlock the full potentials of modern commodity CPUs.

Future Research Directions: Emerging processor architectures have new characteristics and lead to opportunities for further optimization. For example, the 'Many Integrated Core' (MIC) architecture [17] combines a large number cores on a single chip (e.g., Intel Xeon Phi), so that the access time of data items across different cores may depend on the distances between those cores. It becomes important to distribute the workload and transfer data carefully among different cores / threads.

Although our proposed techniques can accelerate existing algorithms by 2-6 times in a single machine, they would take a few hours for very long queries (especially for DTW similarity search). It becomes important to investigate parallel algorithms that run on multiple machines. Some open issues include how to distribute the load among machines, and how to reduce the communication cost among machines.

8 Acknowledgement

This project was supported by grant GRF 152043/15E from the Hong Kong RGC and grant MYRG2014-00106-FST from UMAC Research Committee and grant NSFC 61502548 from National Natural Science Foundation of China.

References

1. Intel 64 and IA-32 architecutres optimization reference manual. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`. [Online; accessed 20-June-2016].
2. Source codes and datasets for experimental study. `http://goo.gl/mwDTxP`. [Online; accessed 20-June-2016].
3. A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *VLDB, Edinburgh, UK*, pages 266–277, 1999.
4. I. Assent, R. Krieger, F. Afschari, and T. Seidl. The ts-tree: efficient time series search and retrieval. In *EDBT*, 2008.
5. V. Athitsos, P. Papapetrou, M. Potamias, G. Kollios, and D. Gunopulos. Approximate embedding-based subsequence matching of time series. In *SIGMOD*, 2008.
6. C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, 2013.

7. S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, 2011.
8. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
9. A. Camerra, T. Palpanas, J. Shieh, and E. J. Keogh. isax 2.0: Indexing and mining one billion time series. In *ICDM*, 2010.
10. S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *TODS*, 32(3):17, 2007.
11. J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *PVLDB*, 1(2):1313–1324, 2008.
12. H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. J. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *PVLDB*, 1(2):1542–1552, 2008.
13. C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, 1994.
14. A. W. Fu, E. J. Keogh, L. Y. H. Lau, C. A. Ratanamahatana, and R. C. Wong. Scaling and time warping in time series querying. *VLDB J.*, 17(4):899–921, 2008.
15. J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.
16. H. Inoue, M. Ohara, and K. Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proceedings of the VLDB Endowment*, 8(3), 2014.
17. S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *PVLDB*, 8(6), 2015.
18. E. Keogh and C. A. Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and information systems*, 7(3):358–386, 2005.
19. Y. Li, L. H. U, M. L. Yiu, and Z. Gong. Discovering longest-lasting correlation in sequence databases. *PVLDB*, 6(14):1666–1677, 2013.
20. A. Mueen, E. J. Keogh, Q. Zhu, S. Cash, and M. B. Westover. Exact discovery of time series motifs. In *SDM*, 2009.
21. P. Papapetrou, V. Athitsos, M. Potamias, G. Kollios, and D. Gunopulos. Embedding-based subsequence matching in time-series databases. *ACM TODS*, 36(3):17, 2011.
22. T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *KDD*, 2012.
23. K. A. Ross. Efficient hash probes on modern processors. In *ICDE*, 2007.
24. D. Sart, A. Mueen, W. A. Najjar, E. J. Keogh, and V. Niennattrakul. Accelerating dynamic time warping subsequence search with gpus and fpgas. In *ICDM*, 2010.
25. J. Shieh and E. J. Keogh. *i*sax: indexing and mining terabyte sized time series. In *KDD*, 2008.
26. A. H. Shoeb and J. V. Guttag. Application of machine learning to epileptic seizure detection. In *ICML*, 2010.
27. S. Sridharan and J. M. Patel. Profiling r on a contemporary processor. *Proceedings of the VLDB Endowment*, 8(2), 2014.
28. L. Xiao, Y. Zheng, W. Tang, G. Yao, and L. Ruan. Parallelizing dynamic time warping algorithm using prefix computations on gpu. In *HPCC/EUC*, 2013.
29. J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *SIGMOD*, 2002.
30. H. Zhu, G. Kollios, and V. Athitsos. A generic framework for efficient and effective subsequence retrieval. *PVLDB*, 5(11):1579–1590, 2012.