OneJoin: Cross-Architecture, Scalable Edit Similarity Join for DNA Data Storage Using oneAPI

Eugenio Marinelli Raja Appuswamy
EURECOM
Biot, France
firstname.lastname@eurecom.fr

ABSTRACT

Synthetic DNA has received much attention recently as an archival storage media due to its high density and durability characteristics. However, the process of retrieving data from DNA is computationally bottlenecked by a key *read consensus* stage that effectively performs an edit similarity join to identify millions of unique consensus strings from hundreds of millions of noisy copies. In this work, we present an end-to-end DNA data decoding pipeline based on *OneJoin*–a cross-architecture edit similarity join that can exploit multicore CPUs, integrated GPUs, and multi-vendor discrete GPUs using a single code base. Central to the effectiveness of OneJoin is the use of oneAPI–an open, standards-based unified programming model for achieving portable data parallelism. Based on a rigorous experimental evaluation using macrobenchmarks and real-world data from DNA storage experiments, we show that OneJoin can provide up to 21× improvement in performance over other state-of-the-art joins and reduce the overall DNA data decoding time from several hours to just a few minutes.

1 INTRODUCTION

The growing adoption of AI and analytics has resulted in the amount of enterprise data growing at a cumulative annual rate of over 40% [20]. Analysts predict that Global Datasphere will expand to 160 Zettabytes by 2025 [20]. However, studies have shown that only 20% of data stored is "hot", or accessed frequently, with the remaining 80% being "cold", or accessed rarely [11, 12, 22]. Cold data has been identified as the fastest growing data segment with a 60% annual growth rate, and also as the segment with the longest lifetime with retention periods of 50–60 years [21]. Unfortunately, current storage media suffer from several limitations that complicates cost-effective archival of cold data [7]. Thus, researchers have started exploring radically new storage media that can offer orders of magnitude improvement in density and durability. One such media that has received a lot of attention recently is Deoxyribo Nucleic Acid (DNA).

Synthetic DNA can be up to eight orders of magnitude denser than tape, and can last several millennia, making it an ideal medium for cold data storage [7]. Using DNA as a digital storage medium requires mapping binary digital data into a quaternary (A,C,G,T) oligonucleotide (oligo) sequence using an encoding algorithm. Once encoded, the oligos are used to synthesize DNA using a chemical process that assembles the DNA one nucleotide at a time. Due to synthesis limitations, each oligo cannot be longer than a few hundred nucleotides. Thus, digital data is stored using millions of oligos. Data stored in DNA is read back by sequencing the DNA molecules which produces several noisy copies of the original oligos, also called reads. As modern sequencers produce millions of reads per run, each oligo is covered by multiple reads. Thus, a read consensus procedure is required to infer the original oligos based on reads so that the inferred sequences can be passed to a decoder to recover the original data.

At its core, the consensus procedure can be viewed as a large-scale string similarity join problem. Given a collection of strings, or reads in our case, the task of similarity join is to find all pairs of strings that are within a threshold distance. As the reads are noisy and can contain insertion, deletion, and substitution errors, it is necessary to use edit (Levenshtein) distance as the comparison metric. Unfortunately, computing exact edit distance between two strings is an intractable problem that does not have a sub-quadratic solution [2]. The DNA data storage scenario amplifies this problem as it requires edit similarity computation across millions of strings. As a result, read consensus is a key computational bottleneck in the DNA data archival pipeline.

Recent work in metric embeddings has led to the development of randomized algorithms that can transform a set of strings into an embedded representation such that the edit distance between two strings in the original set can be approximated by the Hamming distance between strings in the embedded set [5]. EmbedJoin [28] is a new string similarity join that uses edit-to-hamming embedding together with Locality Sensitive Hashing (LSH) [9] to consistently outperform other join algorithms across a range of string lengths and edit distance thresholds. This makes EmbedJoin relevant to the task of read consensus in DNA data storage given the noisy nature of reads. Unfortunately, despite such improvements, EmbedJoin is unable to meet the scalability demands of DNA read consensus due to the sequential nature of algorithm design that limits its execution to a single CPU.

Figure 1 demonstrates this by breaking down the execution time of various stages of EmbedJoin on an input dataset of 470k strings (experimental setup in Section 4). Clearly, EmbedJoin minimizes the overhead of exact edit distance computation, as less than 5% of time spent in the verification phase that performs this computation. However, the total execution time is still nearly 9 minutes for this dataset of just 470k strings, as the filtering stages involving Embedding and LSH are sequential by design to realize a few key optimizations, and they contribute to more than 90% of overall execution time. While a sequential design might be sufficient for small datasets, it makes EmbedJoin unscalable for DNA read consensus with millions of reads.

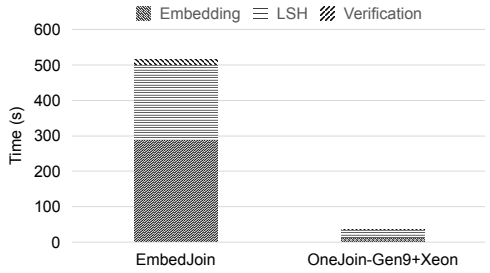


Figure 1: Execution time break down of EmbedJoin and OneJoin under GEN-470KS dataset.

We make three contributions in this work.

- We present OneJoin, a cross-architecture, edit similarity join implemented using oneAPI–an open, standards-based unified programming model for achieving portable data parallelism. In prior work, we developed XJoin, the first database operator to be implemented using oneAPI [16]. OneJoin is the first hardware-accelerated approximate edit similarity join. We describe various design aspects involved in transforming EmbedJoin, a single-threaded, CPU-based algorithm, into OneJoin, a oneAPI-based data parallel algorithm that can execute on CPUs, on-die integrated GPUs, and multi-vendor, PCIe-attached discrete GPUs using a single code base.
- We present a scalable solution to the DNA read consensus problem using OneJoin. Current read consensus solutions are either open-source but non scalable [23], or scalable but properietary [18]. We make our code [1] publicly available to provide a reference implementation for further work on both oneAPI-accelerated data analytics and DNA read consensus.
- Using both synthetic and real-world experimental data, we present an experimental evaluation with the following goals: (i) benchmark the performance of OneJoin with state-of-the-art similarity joins, (ii) demonstrate the benefit of using DPC++ by evaluating OneJoin with heterogeneous processors (CPU and GPU), and (iii) show the scalability of OneJoin by using it for DNA read consensus. Figure 1 shows an example where OneJoin processes the same dataset 14× faster than EmbedJoin by utilizing a 6-core Xeon CPU and an Gen9 Intel iGPU for key data-parallel computations, making it a scalable candidate for DNA read clustering.

2 BACKGROUND

In this section, we first describe CGK Embedding and Locality-Sensitive Hashing for Hamming distance. These are the two main algorithmic tools used by EmbedJoin in performing an edit similarity join. Then, we describe the EmbedJoin algorithm itself.

2.1 Embedding

The edit distance between two strings x and y is defined as the minimum number of insertion, deletion, or substitution operations required to change x into y. Hamming distance, in contrast, only counts the number of substitution operations required to change x into y. For instance, the strings *ACACT* and *GACAC* have a hamming distance of five as none of the characters match, but an edit distance of two, as inserting a G and deleting a T in the first string produces the second string. Thus, compared to Hamming distance, edit distance retains the information of the orderings of characters and captures the best alignment between two strings. However, edit distance is computationally more expensive than Hamming distance; while computing the latter can be done in linear time, computing edit distance requires a dynamic programming formulation with quadratic time complexity [2]. Thus, researchers have investigated metric embedding techniques that can be used to transform strings into an embedded representation making it possible to model problems over those strings in an easier metric space.

CGK embedding is one such algorithm that was proposed recently by Chakraborthy et al.[5]. Applied to the DNA read dataset, the input is a string sequence of length N consisting of four possible characters (A, C, G, T). The output of the CGK embedding algorithm is an embedded string of length $3N$ consisting of the four characters and possibly multiple repeats of a pad character (P). In each iteration, the algorithm appends a character from the input string, or the pad if it runs out of the input string, to the output string. Then, it uses a random binary bit string of length $3N$ to decide if the input index should be advanced. The net effect of this algorithm is that some input characters appear uniquely in the output string, while others are randomly repeated multiple times. Using the theory of simple random walks, Chakraborty et al.[5] established that CGK embedding can embed strings such that Hamming distance of embedded strings is at most square of the edit distance between original strings.

2.2 LSH for Hamming Distance

An important benefit of edit-to-Hamming embedding is that algorithmic tools, like Locality Sensitive Hashing (LSH), that are well defined for Hamming distance but not edit distance, can now be used to identify similar strings without doing an exhaustive pairwise search. Here, we only provide an overview of LSH [9] for hamming distance.

Suppose we have $h_i(p) = p_i$, that is, our hash function is the ith bit of a bit string p of length N. Let $D(p, q)$ be the Hamming distance between bit strings p and q, that is, the number of location-wise different bits between p and q. Then $1 - \frac{D(p,q)}{N}$ is the collision probability, that is the probability that a randomly chosen bit position is the same between the two strings ($h_i(p) = h_i(q)$). Thus, the bit-sampling LSH family for Hamming distance given as

$$\mathcal{H}_N = \{h_i : h_i(b_1 ... b_N) = b_i \mid i \in [N]\}$$

is $\left(d_1, d_2, 1 - \frac{d_1}{N}, 1 - \frac{d_2}{N}\right)$-sensitive for hamming distances $d_1 < d_2$. What this effectively means is that by using random bit-sampling hash functions, it is possible to group strings into buckets such that strings within a bucket are similar to each other in terms of Hamming distance than strings across buckets with a high probability. Given the probabilistic nature of LSH, there will be false positives, where dissimilar strings are grouped together, and false negatives, where similar strings are not identified. The standard AND-OR composition method can be used to reduce the rate of both false positives and negatives. First, m hash functions are concatenated

[1] https://github.com/Eug9/oneoligo.git

in an AND-construction to define

$$f = h_1 \circ h_2 \circ ... \circ h_m, where\ \forall i \in [m], h_i \in_r H$$

such that for $x \in U$ (where U is the set of input strings), $f(x) = (h_1(x)\, h_2(x) ...h_m(x))$ is a vector of m bits. Let $\mathcal{F}(m)$ be the set of all hash functions f. Then, z such functions are grouped in an OR-construction to define

$$g = f_1 \vee f_2 \vee ... \vee f_z, where\ \forall j \in [z], f_j \in_r \mathcal{F}(m)$$

such that for $x, y \in U, g(x) = g(y)$ if and only if there is at least one $j \in [z]$ for which $f_j(x) = f_j(y)$. g has been shown to be $\left(d_1, d_2, 1-(1-(d_1/N)^m)^z, 1-(1-(d_2/N)^m)^z\right)$ sensitive, and carefully selecting m and z can amplify the gap between p_1 and p_2 and reduce both false positives and false negatives.

2.3 EmbedJoin

EmbedJoin is a string similarity join algorithm that builds on CGK-Embedding and LSH to find all pairs of strings whose edit distance is within a given threshold K. The core algorithm consists of two phases, namely, *filtering* and *verification*.

During filtering, EmbedJoin first sorts the input strings based on their length. Then, it processes one input string at a time by first embedding the string s_i into t_i using CGK embedding. Then, each hash function f_j is used to hash the embedded string to a bucket $B = f_j(t_i)$. For every other string s_b found in B, a length comparison is performed. If the difference in length is less than edit threshold, $< s_i, s_b >$ is considered a potential pair and added to a candidate list C. However, if the difference is more than the threshold, then it follows that $ED(s_b, s_j) > K\ \forall j > i$ due to the fact that input strings are sorted based on length. Thus, EmbedJoin deletes s_b from bucket B. s_i is itself also stored in bucket B for future comparisons. As each pair $< s_i, s_j >$ can potentially be added into C multiple times by different LSH hashes, a duplicate elimination is performed. This process is repeated for each input string, and at the end of filtering stage, EmbedJoin accumulates a list of all possible candidate pairs. During the verification stage, it iterates over these pairs, computes the exact edit distance, and adds pairs with edit distance below the user-specified threshold K to the output set.

So far, we presented the vanilla EmbedJoin algorithm where each input string is embedded only once. However, the randomization in CGK embedding can occasionally fail to provide a low-distortion mapping from edit to hamming distance when certain bit patterns are used in randomization. As a guard against potential errors that can be introduced by such poor embedding, the authors extended EmbedJoin by embedding each string multiple times using different random seeds. The resulting algorithm was referred to as EmbedJoin+ [28]. We would like to point out here that we use EmbedJoin+ as the baseline for implementation and evaluation although we refer to it as EmbedJoin in the rest of this paper.

EmbedJoin demonstrated that embedding–LSH combination can eliminate the need to perform $O(N^2)$ pairwise edit distance computations. However, the current algorithm is designed to use sequential filtering to (i) simplify the logic of updating and searching hash buckets during LSH, and (ii) incorporate the sort-based bucket-trimming optimization described above. Due to this reason, the filtering stage contributes to over 90% of overall execution time (Figure 1).

3 DESIGN AND IMPLEMENTATION

Modern server hardware is increasingly heterogeneous with a diverse mix of scalar, vector, matrix, and spatial architectures deployed in CPU, GPU, FPGA, and other accelerators. OneJoin is a data-parallel redesign of the sequential EmbedJoin algorithm in order to fully exploit such heterogeneous parallelism. Central to the redesign is the use of oneAPI–a cross-industry effort for developing an open, standards-based unified programming model to simplify software development across diverse accelerator architectures.

Conceptually, OneJoin follows the same high-level approach as EmbedJoin as it also consists of a filtering stage followed by a verification stage. However, unlike EmbedJoin, the OneJoin filtering stage is internally organized as a collection of several data-parallel kernels programmed in Data Parallel C++ (DPC++). In contrast to other data-parallel languages that are proprietary (CUDA) or low level in nature (OpenCL), DPC++ is an open-source implementation of SYCL–an industry-wide standardization effort to define cross-platform data parallelism support for standard C++. DPC++ builds on Clang and LLVM compilers to support key data-parallel constructs defined in the SYCL standard, and in doing so, provides a cross-platform abstraction layer for data parallelism.

In the rest of this section, we first describe the various data-parallel stages of OneJoin. Then, we detail the cross-architecture fork–join execution model used by OneJoin to simultaneously exploit CPUs and GPUs. Finally, we describe the end-to-end DNA data decoding pipeline using OneJoin. We will provide an overview of key aspects of DPC++ when relevant. We refer the reader to related resources for further details about DPC++, oneAPI and a comparison to other accelerator programming models [19].

3.1 Data-Parallel Edit Similarity with OneJoin

3.1.1 Data-parallel Embedding. The embedding stage is the first data-parallel stage in OneJoin. The data-parallel embedding in OneJoin is based on the observation that while embedding of a single input string is not parallelizable due to data dependencies (each embedded character depends on the previous one and the random bit string), embeddings of different input strings, and even embeddings of even same input string but with different random bit strings, are parallelizable. Thus, OneJoin implements embedding as a data-parallel kernel such that each kernel instance performs a CGK embedding for one specific $\{input\_string, random\_string\}$ combination.

As accelerators often have on-board memory that is different from the system memory attached to the CPU, DPC++ provides the `buffer` abstraction to manage memory. `Buffer` objects can be created from existing data on the host, and in such a case, data is automatically copied from the host address into the buffer. When the buffer is used in a kernel, the runtime automatically ensures that the underlying data is made available to the kernel by coping it from host memory to device memory. OneJoin uses DPC++ buffer to store both input strings and embedded output strings. Doing so ensures that the runtime will automatically make the data available to the embedding kernel irrespective of the device on which it is executed. However the size of a single buffer can be limited depending on the platform and processor used for kernel execution.

OneJoin overcomes this limitation by partitioning input strings into batches, where the batch size is empirically chosen such that all input and output data within a batch can fit in a buffer. Within a batch, OneJoin uses a structure-of-arrays (SoA) organization for its input and output data structures. The characters of the input strings are stored in a flattened 1D input array that is used to create an input buffer. The length of each string is stored separately in an auxiliary array that is used to create an index buffer. With such an organization, the first r kernel instances embed the first input string using different random strings. The next r kernel instances embed the second input string, and so on. Thus, multiple kernel instances will be able to share the result of single memory fetch operation and benefit from sequential prefetching and caching.

On the output front, recall that the embedded string is used as input to LSH. In the AND construction of the m-bit Hamming LSH, m positions are sampled by each hash function, and the hash bucket is determined using the characters at these positions. In the OR construction, there are z such hash functions. Therefore, for each embedded string, OneJoin only generates the $z \times m$ characters required for LSH and stores them in a flattened array that is used to create an output buffer.

Once all the arrays and associated buffers required for processing one batch of strings are allocated and initialized, OneJoin submits the data-parallel embedding kernel for execution. As kernel submission is asynchronous, control is immediately returned back to the host code which prepares the next batch of input strings and submits it asynchronously. Thus, OneJoin overlaps I/O and input processing with embedding.

3.1.2 Data-parallel LSH.. OneJoin decomposes the sequential LSH filtering stage in EmbedJoin into two data-parallel stages, namely, bucketization, and candidate generation.

Bucketization. The embedding stage converts each input string into r embedded strings. The role of LSH filtering is to hash embedded strings into buckets such that similar strings are grouped together in a bucket. Two strings end up in the same bucket if they are embedded using the same random bit string and have at least one hash function produce identical hash ID. EmbedJoin computes the hash ID using a standard two-level hashing implementation of LSH [28]. First, the m characters corresponding to a hash function are first converted into a vector u of ASCII-equivalent integers. Then a random vector $v \in 0, 1..., P-1$ is used to compute the hash ID as $< u.v > mod\ P$, where $u.v$ denotes the inner product, and P denotes a large prime number. OneJoin implements hash ID computation as a data-parallel kernel, where each kernel instance performs one hash computation using m characters.

Once all hash IDs have been computed, the next step is to group strings into buckets. One way to achieve this, as done by the original EmbedJoin, is to physically create $r \times z$ distinct hash tables, and use the hash IDs generated earlier to identify buckets in each hash table. Two strings would thus end up in the same bucket of a hash table only if they are embedded using the same random bit string, and at least one hash function produces identical hash ID. However, such an approach complicates parallelization as multiple threads need to synchronize their updates to the hash table. Instead, observe that once a hash ID has been computed, any hash bucket can be uniquely identified by a three-tuple $< t, k, id >$, where $t \in [0, r]$ is an index that represents the random string used, $k \in [0, z]$ is an index that represents the hash function used, and id is the hash ID computed. Thus to bucketize strings, we first modify the hash ID computation stage so that the output is organized as an array of four-tuples $< t, k, id, i >$, where t, k, id are defined above, and $i \in [0, N]$ represents the index of the corresponding input string whose embedded characters were used in the hash computation. Once hash ID computation is done, buckets are identified by simply sorting the output array of four-tuples, all identical $< t, k, id >$ tuples identifying a bucket are adjacent after sorting.

Candidate generation. Due to the use of two-level LSH for hash ID computation, a hash function can wrongly map two strings to the same bucket in some cases even if all characters used by that hash function do not match. Thus, it is necessary to perform a validation to ensure that the embedded characters used for hash computation from the two strings are indeed identical before marking them as candidates. There are several problems in parallelizing this validation step with OneJoin.

First, in the original EmbedJoin, the use of physical hash tables provided the benefit that once a bucket was identified for a given string, all other strings found in that bucket can immediately be identified as candidates for verification. However, in OneJoin, we do not maintain an explicit hash table. Thus, we need first derive the list of potential candidates from the array of five tuples. Second, different buckets can produce a different number of candidates, and there is no way to predict this a priori. This makes it difficult to determine the amount of memory to allocate for storing candidate pairs, and identify the optimal granularity of parallelization.

To solve these problems, we decompose this step into a series of data-parallel computations. First, we determine the total number of buckets B and the number of strings per bucket in the five-tuple array. Using the counts, we then compute the total number of all possible candidate pairs $C = \Sigma_{b=1}^{B} \frac{n_b \times (n_b - 1)}{2}$, where n_b denotes the number of candidates per bucket, and preallocate memory to hold C candidate pairs. Finally, we implement the validation as a data-parallel kernel such that each kernel instance performs a comparison for one candidate pair. The outcome of these three steps is an array that contains for each possible candidate pair the status as to whether LSH filtering succeeded or failed. We finally use this output array to deduplicate and extract unique candidates that are passed along for exact edit distance-based verification.

3.1.3 Cross-architecture fork–join execution. Similar to other accelerator programming approaches, the DPC++ platform model makes the distinction between a *host*, which is typically a CPU, and multiple *devices*, which are accelerators like GPU. The host code is responsible for controlling and coordinating kernel execution on devices. DPC++ provides the abstraction of a *queue* to enable host code to command devices. A queue can be bound to a device of a specific type using a *device selector*.

As described above, the OneJoin filtering stage is composed of three key data parallel kernels that are responsible for embedding, bucketization, and candidate generation. Each of the data-parallel kernels can be executed on a CPU, GPU, or both, by simply using appropriate device selectors in DPC++. OneJoin exploits this portable parallelism by using a cross-architecture fork–join execution method. Each data-parallel kernel is simultaneously launched

on all available processor types, and thus, forms the fork stages of execution. Steps between kernel invocations that handle memory management and data structure reorganization are purely done on the CPU and form the join stages of execution.

In order to facilitate cross-architecture forking, OneJoin uses DPC++ buffers to wrap all kernel-accessible data structures to ensure that the runtime automatically makes the data available to kernels irrespective of where they execute. However, the kernel execution time varies depending on whether it is executed on a CPU or GPU. As we wanted OneJoin to automatically determine the list of available processors at runtime and divide work accordingly, we adopted a simple sampling-based cost estimation strategy. Before each fork stage, OneJoin launches the corresponding kernel on all available devices using a small data sample (1% of the kernel's input data) and measures complete kernel execution time. OneJoin then allocates buffers to each processor at a rate inversely proportional to the profiled execution time, with faster processors being assigned more buffers, so that all processors complete the fork stage at roughly the same time. We found that this simple strategy provides a balanced division of work across CPUs and GPUs without creating stragglers, thanks to the predictable, uniform execution time of kernel instances in all our three kernels.

In addition to the kernels, the filtering stage also contains data-parallel operations like sorting and duplicate elimination. These operations are implemented by calling parallel algorithms in the oneAPI DPC++ Library (oneDPL).

3.1.4 Edit distance verification. Once all the candidate pairs that survive the LSH filtering have been generated, the final stage in OneJoin is verification by means of the exact edit distance computation. Given that verification contributes to less than 5% of overall processing time under EmbedJoin (Figure 1), we parallelize this stage only on CPUs by simply partitioning the list of candidates across several CPU threads, and having each thread independently performs verification of a subset of candidate pairs.

3.2 Read Consensus with OneJoin

Having described OneJoin, we will now present the design of our end-to-end DNA data restoration pipeline with OneJoin. The first step in restoring data from DNA is sequencing the DNA to generate reads that are noisy copies of the original oligos. The second step involves passing these reads to OneJoin which identifies all possible pairs of similar reads. In the third step, the result from OneJoin is used as input to a clustering stage. We use the well-known density-based DBSCAN [8] algorithm to cluster the reads. At the core of DBSCAN is a range query responsible for retrieving the ϵ-neighborhood of a point. Dynamically executing the range query for each point would require a repeated edit similarity search to identify all possible points that are similar to a given query point. As OneJoin precisely provides this result for each point, we simply use the precomputed output from OneJoin as a substitute for the range query without having to actually execute the query. Once the clusters are identified by DBSCAN, we iterate over each cluster and perform a position-wise consensus to build a single representative oligo from each cluster. Finally, these inferred oligos are passed to the decoding algorithm that converts the quaternary code back into binary data.

Dataset	n	Avg. Len	$\|\Sigma\|$
Join datasets			
TREC	233,435	1217	37
UNIREF	400,000	445	25
GEN-470KS	470,492	5000	4
DNA storage datasets			
Synthetic	5M,10M,20M	209	4
Real	19M	91	4

Table 1: Parameters of datasets used in this work.

4 EVALUATION

In this section, we will present the experimental results. The evaluation is structured as follows. First, we will demonstrate the benefit of using oneAPI by demonstrating the portability of OneJoin across multiple processor types (Sections 4.2–4.4). Then, we will present benchmarks comparing OneJoin to EmbedJoin and other popular join algorithms under several publicly available datasets (Section 4.5). Finally, we will present results from an end-to-end DNA data decoding experiment and demonstrate the benefit of OneJoin for read consensus (Section 4.6).

4.1 Experimental Setup

Hardware Setup. All experiments are conducted on either the Intel® DevCloud using a compute instance with a 6-core Xeon® E-2176G CPU clocked at 3.7GHz, 64GB DRAM, and a Gen9 Intel® iGPU, or on a local server equipped with a 12-core Intel® Core i9-10920X CPU clocked at 3.5GHz, 128GB DRAM, and a NVIDIA GeForce RTX 2080 Ti dGPU.

Software Setup. OneJoin is implemented in DPC++ and compiled using DPCPP (O3). We compare OneJoin with EmbedJoin, AdaptJoin[25], and QChunk[17]. We choose these algorithms as they have been demonstrated to be the best alternatives for edit similarity joins [28]. We also compare the OneJoin-based DNA read consensus approach with Starcode [23], a widely-used read clustering program we used previously for consensus.

Datasets. For comparing OneJoin with other join algorithms, we use three publicly-available, real-world datasets similar to the original EmbedJoin paper [28]. Table 1 summarizes the key characteristics of each dataset. Further details about the DNA storage dataset are provided in Section 4.6.

UNIREF. A dataset of UniRef90 protein sequence data from UniProt project. Each sequence is an array of amino acids in upper case. We remove sequences whose lengths are smaller than 200, and then extract the first 400,000 protein sequences similar to prior studies [28].

TREC. A dataset of references from Medline database consisting of titles and abstracts from 270 medical journals.

GENOMICS. Dataset with 470k strings based on Chromosome 20 of 50 individuals obtained from the personal genomes project. The long DNA sequences are partitioned into shorter substrings of length 5,000.

4.2 Portable Parallelism

In this section we demonstrate the utility of DPC++ in providing portable parallelism by providing a comparison between two variants of OneJoin, namely, *OneJoin-DPL-Xeon* which uses only a

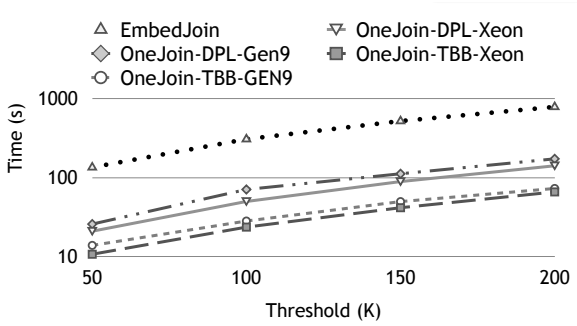


Figure 2: Exec. time of EmbedJoin and OneJoin under GEN-470KS dataset at various edit distance thresholds.

multicore CPU, and *OneJoin-DPL-GEN9* which uses only an integrated Intel® Gen9 GPU. The results in this section are obtained from the Intel® DevCloud server. Figure 2 shows the execution time of OneJoin and EmbedJoin for different edit distance thresholds under GEN-470KS, our largest join dataset. We verified that the result produced by OneJoin is identical to EmbedJoin in all cases. The other join algorithms either failed to execute, or took longer than 6 hours, as they were unable to handle the GEN-470KS dataset. Results under TREC and UNIREF datasets are reported in Section 4.5.

As shown in Figure 2, OneJoin-DPL-Xeon provides a $5.5 - 6.5\times$ speedup compared to EmbedJoin. Similarly, OneJoin-DPL-GEN9 uses the iGPU to provide a $4.5 - 5.5\times$ speedup respectively compared to EmbedJoin. The only difference in code between CPU and GPU versions of OneJoin is the use of a different device selector (`gpu_selector` versus `cpu_selector`), as the same kernel code is compiled to different processor architectures. This result clearly highlights the benefit of DPC++ in achieving portable parallelism across different processor architectures.

Recall that both OneJoin and EmbedJoin consist of embedding, LSH, and verification stages. In order to understand the relative improvement in performance of each stage, we report the per-stage execution time of EmbedJoin and OneJoin in Figure 3 while fixing the edit distance value at 150. Recall that unlike embedding and LSH, OneJoin always performs verification on the CPU in a multi-threaded fashion. This is why Figure 3 has no GEN9 configuration for the verification stage of OneJoin. It can be seen that there is a big difference between the embedding and LSH stages. OneJoin achieves a 12× reduction in embedding time using the iGPU, and a 20× reduction with the 6-core CPU over EmbedJoin. However, it only achieves a 2.41× reduction in LSH time with the iGPU, and 2.79× reduction with the 6-core CPU.

Recall that the embedding stage in OneJoin is a single data-parallel kernel, while the LSH stage is implemented using multiple kernels with intervening calls to oneDPL library. oneDPL enables callers to pick the processor used for executing a library call by changing a device-selection-policy parameter. Thus, the OneJoin-DPL-GEN9 configuration invokes library implementations optimized for the iGPU, while OneJoin-DPL-Xeon invokes CPU-optimized implementations. Despite this, benchmarking revealed that while the LSH kernels scaled well and contributed to only 10% of total LSH execution time, calls to oneDPL accounting for the remaining 90% under both CPU and GPU. On further investigation, we found that oneDPL implementation of various tasks like sorting and reduction were inferior to those provided by the purely-CPU-based parallel algorithm library in Intel® Thread Building Blocks (oneTBB). So we developed a new version of OneJoin by replacing oneDPL with oneTBB. Note that by doing so, we lose the benefit of being able to execute parallel algorithms on the GPU; the data-parallel kernels written in DPC++ which form the "fork" stages of OneJoin can still work on either CPU or GPU, but calls to oneTBB during the "join" stages work only on the CPU.

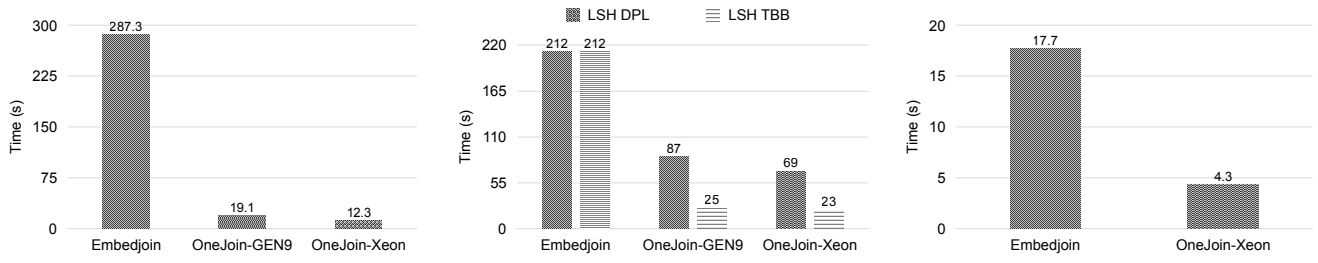
Figure 3b presents a comparison of the execution time of the LSH stage for both versions of OneJoin (oneDPL and oneTBB). Clearly, OneJoin performs better with oneTBB than oneDPL, as OneJoin-TBB-Xeon provides a 9× speedup, and OneJoin-TBB-GEN9 provides a 8× speedup over EmbedJoin for the LSH stage. Now, returning back to Figure 2, we see that this improvement directly translates into a reduction in overall execution time, as OneJoin-TBB-GEN9 achieves a 10.5× speedup over EmbedJoin (versus 4.5× with OneJoin-DPL-GEN9), and OneJoin-TBB-Xeon achieves a 12.5× speedup over EmbedJoin (versus 5.5× with OneJoin-DPL-Xeon). Given this difference, all the results reported henceforth for OneJoin are based on oneTBB.

4.3 Cross-architecture Fork

One of the design aspects of OneJoin we described in Section 3.1.3 is cross-architecture fork, or its ability to run data-parallel kernels simultaneously on multiple processors. Figure 4a shows the isolated execution time of just the embedding kernel in four configurations of OneJoin under the GEN-470KS dataset while fixing the edit distance threshold at 150. *Xeon-1C* performs embedding on a single CPU core. *Xeon* uses all 6 CPU cores. *GEN9* performs embedding on the iGPU. The results for Xeon and GEN9 are similar to those shown in Figure 2. Finally, in *Xeon+GEN9*, the embedding kernel is executed on both 6-core-CPU and iGPU, with work allocation being performed on the fly using our sampling-based cost estimation method.

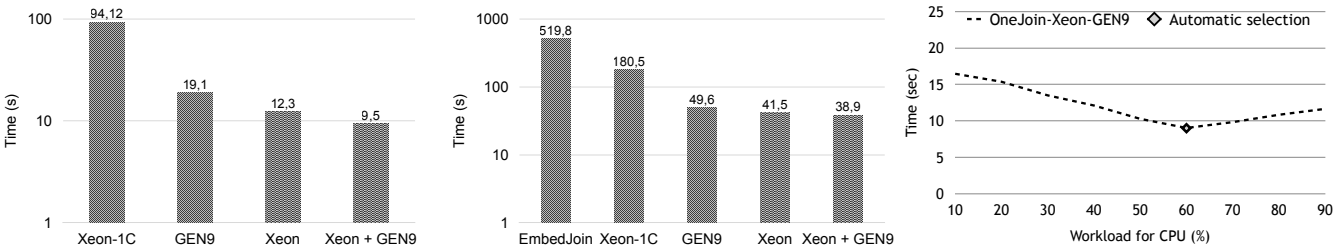
Comparing single-threaded Xeon-1C in Figure 4a with equivalent EmbedJoin in Figure 3a, we see that our optimized code base accounts for a 3× reduction in embedding time. Comparing one-core (Xeon-1C) with 6-core (Xeon) and GEN9 results in Figure 4a, we see that oneAPI effectively parallelizes the embedding kernel across multicore CPUs and iGPU. Finally, looking at Xeon+GEN9 result, we see the benefit of cross-architecture fork, as OneJoin-Xeon+GEN9 is 1.3×/2× faster than its CPU-only/GPU-only counterparts.

Figure 4b shows the total execution time of EmbedJoin and OneJoin under the same four configurations and benchmark. Similar to Figure 4a, we see that utilizing both devices simultaneously widens the gap between OneJoin and EmbedJoin, as OneJoin-Xeon+GEN9 provides a 14× speedup over EmbedJoin compared to the 11× speedup achieved by OneJoin-GEN9, and 13× speedup achieved by OneJoin-Xeon. However, comparing Figure 4a and Figure 4b, we see that improvements obtained for the embedding kernel do not get translated into a corresponding reduction in overall execution time. There are two reasons for this. First, cross-architecture fork only optimizes the fork stages of OneJoin (kernels in embedding



(a) Embedding Phase (b) LSH Phase (c) Verification Phase

Figure 3: Execution time breakdown of the three phases of OneJoin/EmbedJoin



(a) Cross-arch. fork: Embedding time. (b) Cross-arch. fork: Total time. (c) Sensitivity to workload division.

Figure 4: Cross-architecture fork evaluation.

and LSH). The join stages between data-parallel kernels based on oneTBB, and the verification phases of OneJoin run entirely on the CPU and do not benefit from cross-architecture fork. Second, the Intel® Xeon CPU is much faster than the GEN9 iGPU as shown by Figure 4a. As a result, data-parallel kernels running on the CPU are assigned more data by our sampling-based work allocator to ensure that the CPU and GPU are kept fully utilized.

To isolate the impact of work allocator, Figure 4c shows the execution time of the embedding stage where we manually vary work allocation. The dark circle shows the allocation automatically picked up our allocator. Clearly, a balanced work allocation can provide up to 50% reduction in time and our sampling-based allocator provides an optimal division of work. These cross-architecture fork experiments demonstrate that DPC++ enables the execution of a single-source data parallel kernel simultaneously on multiple processor types.

4.4 Cross-Platform Parallelism

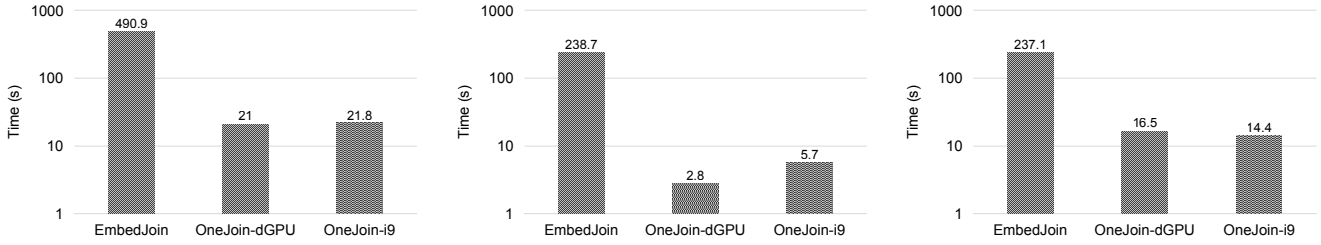
So far, we demonstrated the performance of OneJoin using Intel® iGPU and CPU. In this section, we will present results from our local server that is equipped with a 12-core CPU and a PCIe-atached, NVIDIA dGPU. In order to run oneAPI on NVIDIA GPU, we used CodePlay's SYCL-for-CUDA extension that allows compiling applications written in DPC++ to run on NVIDIA dGPUs. In terms of code, the main change required is the recompilation of OneJoin with a modified Clang++–LLVM compilation infrastructure that supports a CUDA backend.

Figure 5a shows the execution time of EmbedJoin and OneJoin using the 12-core CPU or the NVIDIA dGPU for executing kernels under the GEN-470KS dataset. While data-parallel kernels run on the CPU or dGPU depending on the configuration in the fork stages, note that we use the CPU-based Intel oneTBB in both cases for executing various parallel algorithms in the join stages. Due to limitations in the compilation infrastructure, it is not possible to achieve cross-architecture fork across Intel® CPUs and NVIDIA discrete GPUs as on date. Thus, we are unable to report results for a mixed execution.

As can be seen in Figure 5a, OneJoin-dGPU provides a 21× speedup over EmbedJoin. We can also see that OneJoin-dGPU provides performance comparable with the OneJoin-i9 despite the fact that the CPU has 12 cores. Comparing this with the results from Figure 4b, we see that dGPUs are much more effective than iGPUs despite added overheads, like PCIe data transfers. To understand the per-stage contribution to overall improvement, Figure 5b,5c show the breakdown across embedding and LSH stages. We do not show verification stage as it is executed entirely on the CPU and provides a linear 12× speedup as expected. As shown by the breakdown, the majority of the 21× improvement stems from the reduction in embedding time, as OneJoin executes embedding 94× faster than EmbedJoin with the NVIDIA dGPU, and 45× faster with the 12-core CPU. In contrast, the LSH stage only benefits from 11–12× improvement due to the use of CPU-based oneTBB library for parallel algorithms. Given that no code change was required to get OneJoin to run on the NVIDIA dGPU, this result demonstrates the cross-platform portability of DPC++.

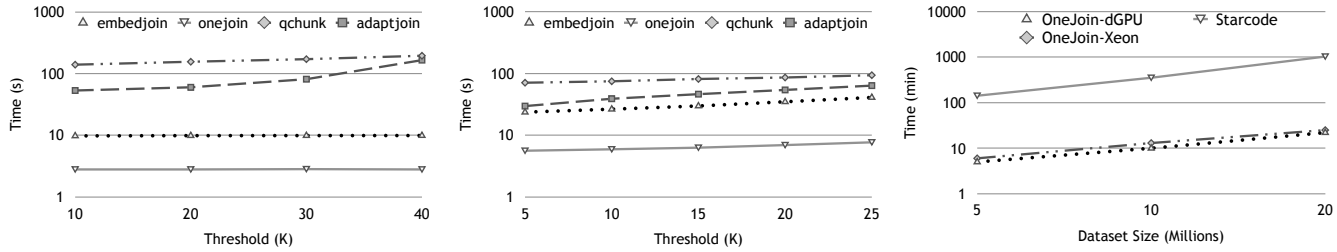
4.5 Comparison with State-of-the-art Joins

In this section, we will present macrobenchmarks to compare OneJoin to other state-of-the-art join algorithms under other join datasets. All results reported in this section are obtained using our



(a) Total execution time. **(b) Embedding execution time.** **(c) LSH Execution time.**

Figure 5: Execution time of OneJoin with discrete GPUs.



(a) TREC **(b) UNIREF** **(c) OneJoin-based clustering vs Starcode.**

Figure 6: (a,b) Execution time of join algorithms at various distance thresholds (K); (c) Onejoin versus starcode

local server and we use OneJoin configured to use the NVIDIA GPU as the comparison target. Figures 6a,6b show the execution time of various join algorithms under the other two join datasets described in Section 4.1 as we vary the edit distance threshold. There are two important observations to be made. First, there is clear difference between embedding-based (EmbedJoin, OneJoin) and exact edit distance approaches (QChunk, AdaptJoin), especially at high edit distance thresholds, as EmbedJoin and OneJoin consistently outperform other algorithms under all benchmarks. This demonstrates the potential of embedding and LSH to reduce the overhead of exact edit distance computation, and validates the results obtained by prior work on EmbedJoin [28]. Second, OneJoin provides up to a 3.5×/5.3× speedup over EmbedJoin under TREC/UNIREF. This improvement is lower than those reported under GEN-470KS dataset due to the fact that although these datasets push other join implementations to their scalability limits, they are not intensive enough for OneJoin as the execution time of OneJoin under these datasets is very short. OneJoin can handle much larger datasets and we only report these results for completeness.

4.6 OneJoin for DNA Data Storage

Having separately evaluated OneJoin, we will now demonstrate its utility for the task of read consensus in DNA data storage. We will use two datasets, one simulated, and one obtained from real DNA sequencing, to evaluate our OneJoin-driven read consensus solution.

We generated the simulated dataset by loading 1MB TPC-H data into a postgreSQL database. We used a data archival tool developed in prior work to archive the database and encode it to generate 505,783 oligos, each with a length of 209 nucleotides. We then used BBMap, a read simulator, to generate 5M, 10M, and 20M reads from these original oligos using the Illumina error model that inserts various substitution, insertion, and deletion errors in the reads.

	Data	OneJoin(%)	Starcode(%)
OneJoin	5M	98.2	97.8
Starcode	10M	99.7	99.9
Starcode	20M	99.9	99.9

Table 2: Clustering accuracy of OneJoin and Starcode.

Figures 6c reports wall-clock execution time for our solution and Starcode on our local server, with Starcode running on all 12 cores, and OneJoin using the NVIDIA GPU for the fork stages and 12-core CPU for oneTBB-based join stages. OneJoin-accelerated clustering is 28×–46× faster than Starcode as dataset size increases from 5M to 20M. As the read coverage or the number of original oligos increases, the overhead of edit distance computation in Starcode also increases rapidly. Our solution, in contrast, substantially reduces the number of edit-distance comparisons to identify similar reads due to embedding and LSH, and is capable of exploiting cross-architecture parallelism due to the use of oneAPI. The accuracy shown in Table 2 computed as $\frac{number-of-oligos-detected-via-consensus}{number-of-original-oligos}$, shows that OneJoin is competitive with state-of-the-art in terms of accuracy.

In order to show that our solution is capable of fully recovering data in a real-world setting, we use a dataset obtained from a previous published experimental study that encoded a 12KB postgreSQL database to generate 404 oligos, synthesized DNA, and sequenced it back using Illumina Novoseq 500 sequencer generating 19 million reads, each of which is 91 nucleotides long. We used the Onejoin-based read consensus solution to recover 403 oligos from the noisy

reads. These recovered oligos matched the original oligos exactly. On further inspection we found that the one missing oligo was covered by only one read. As a result the clustering algorithm classified it as a noise point and dropped it. In spite of this, the decoder was able to recover back the original database successfully due to the use of repetition code during encoding. In terms of the execution time, OneJoin dominated overall execution time, as it took 9 minutes to compute all similar pairs. Compared to this, the consensus stage and the decoding stage complete in a few seconds, thanks to the precomputed range query results produced by OneJoin.

To summarize, these results indicate that our OneJoin-based read consensus solution enables end-to-end data decoding for DNA storage in minutes instead of several hours on a single server, thanks to the portable, cross-architecture parallelism provided by DPC++.

5 RELATED WORK

Edit similarity join is a well-known and computationally intensive problem. Several solutions have been designed based on a signature-based approach to perform edit similarity join. AdaptJoin [25], QChunk [17], VChunk [26], GramCount [10], PassJoin [15], ED-Join [27], AllPair [3], FastSS [4], ListMerger [14] are examples of this approach that consists of generating all substrings of a certain length and filtering them based on frequencies, positions and contents. All these algorithms suffer from fundamental design limitations that prevent their scalability under large edit distance thresholds. Alternative approaches are represented by the tree-based algorithm such as M-Tree [6], enumeration-based algorithm such as PartEnum [1], and trie-based algorithm such as TrieJoin [24]. The main issue related to these algorithms is the fact that these are not efficient for large datasets [13]. The DNA read consensus task amplifies the scalability issues faced by these algorithms due to the sheer scale of the dataset.

The closest to our work is the solution developed by Rashtchain et al. [18] that proposes a new clustering algorithm specifically for DNA storage. We are unable to directly compare our solution with this work as this work is closed source. In addition to being open source, our solution also has broader applicability, as OneJoin in itself can be used for further research on edit similarity joins, and our oneAPI implementation can be used for further research on portable parallelism across heterogeneous hardware.

6 CONCLUSION

In this work, we highlighted the scalability issues in computing edit similarity joins over large datasets in the context of DNA data storage. We present OneJoin, a data-parallel join algorithm built using DPC++ and oneAPI. We demonstrated that OneJoin can efficiently exploit heterogeneous parallelism available on modern hardware to provide up to 21× reduction in execution time compared to EmbedJoin (Section 4.4). Using synthetic and real-world DNA storage datasets, we also demonstrated that a OneJoin-enabled read consensus solution can provide up to 46× reduction in execution time, and thus, enable end-to-end data decoding in minutes using a commodity server. We are making OneJoin source code [2] publicly available to encourage further work on using oneAPI and DPC++ for developing cross-platform, cross-architecture operators for data analytics engines.

We are exploring several avenues of future work. On the implementation front, we are investigating the use of more advanced DPC++ features like Unified Shared Memory and nd_ranges for further optimizing performance. We are also investigating the use of CUDA interop to invoke CUDA library calls from DPC++. Doing so will make it possible to move execution of parallel algorithms in the join stages of OneJoin from CPU to NVIDIA dGPUs. On the hardware front, we are extending our evaluation to cover new GPUs, like the recently announced Intel® DG1, and our design to cover spatial architectures (FPGA). On the application front, we are investigating the utility of OneJoin in achieving read consensus with long-read sequencers, like Oxford Nanopore.

[2] https://github.com/Eug9/oneoligo.git

7 ACKNOWLEDGMENTS

This work was partially funded by the European Union's Horizon 2020 research and innovation programme, project OligoArchive, under grant agreement No 863320.

REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. 2006. Efficient exact set-similarity joins. In *VLDB*.
[2] Arturs Backurs and Piotr Indyk. 2015. Edit Distance Cannot Be Computed in Strongly Subquadratic Time (Unless SETH is False). In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*. 51–58.
[3] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up All Pairs Similarity Search. In *Proceedings of the 16th International Conference on World Wide Web* (Banff, Alberta, Canada) *(WWW '07)*. Association for Computing Machinery, New York, NY, USA, 131–140. https://doi.org/10.1145/1242572.1242591
[4] T. Bocek. 2007. Fast Similarity Search in Large Dictionaries.
[5] Diptarka Chakraborty, Elazar Goldenberg, and Michal Kouckỳ. 2016. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*. 712–725.
[6] P. Ciaccia, M. Patella, and P. Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB*.
[7] Semiconductor Research Corporation. 2018. 2018 Semiconductor Synthetic Biology Roadmap. https://www.src.org/program/grc/semisynbio/ssb-roadmap-2018-1st-edition_e1004.pdf.
[8] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*. AAAI Press, 226–231.
[9] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. 518–529.
[10] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. 2001. Approximate string joins in a database (almost) for free. In *VLDB 2001 - Proceedings of 27th International Conference on Very Large Data Bases (VLDB 2001 - Proceedings of 27th International Conference on Very Large Data Bases)*, Peter M. G. Apers, Paolo Atzeni, Richard T. Snodgrass, Stefano Ceri, Kotagiri Ramamohanarao, and Stefano Paraboschi (Eds.). Morgan Kaufmann, 491–500. 27th International Conference on Very Large Data Bases, VLDB 2001 ; Conference date: 11-09-2001 Through 14-09-2001.
[11] IDC. 2013. Technology Assessment: Cold Storage Is Hot Again — Finding the Frost Point. http://www.idc.com/getdoc.jsp?containerId=246732.
[12] Intel. [n.d.]. Cold Storage in the Cloud: Trends, Challenges, and Solutions. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cold-storage-atom-xeon-paper.pdf.
[13] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2014. String Similarity Joins: An Experimental Evaluation. *Proc. VLDB Endow.* 7, 8 (April 2014), 625–636. https://doi.org/10.14778/2732296.2732299
[14] Chen Li, Jiaheng Lu, and Yiming Lu. 2008. Efficient Merging and Filtering Algorithms for Approximate String Searches. *2008 IEEE 24th International Conference on Data Engineering* (2008), 257–266.
[15] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. 2011. Pass-Join: A Partition-Based Method for Similarity Joins. *Proc. VLDB Endow.* 5, 3 (Nov. 2011), 253–264. https://doi.org/10.14778/2078331.2078340

[16] Eugenio Marinelli and Raja Appuswamy. 2021. XJoin: Portable, Parallel Hash Join across Diverse XPU Architectures with OneAPI. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN)*. Article 11, 5 pages. https://doi.org/10.1145/3465998.3466012

[17] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. 2011. Efficient Exact Edit Similarity Query Processing with the Asymmetric Signature Scheme. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 1033–1044. https://doi.org/10.1145/1989323.1989431

[18] Cyrus Rashtchian, Konstantin Makarychev, Miklos Racz, Siena Ang, Djordje Jevdjic, Sergey Yekhanin, Luis Ceze, and Karin Strauss. 2017. Clustering Billions of Reads for DNA Data Storage. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc., 3360–3371. https://proceedings.neurips.cc/paper/2017/file/ab7314887865c4265e896c6e209d1cd6-Paper.pdf

[19] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. 2020. *Data Parallel C++* (1st ed.). Apress Open.

[20] David Reinsel, John Gantz, and John Rydning. 2017. Data Age 2025: The Evolution of Data to Life-Critical. https://www.seagate.com/files/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf. (2017).

[21] SNIA. 2017. 100 Year Archive Requirements Survey 10 Years Later. https://www.snia.org/sites/default/files/SDC/2018/presentations/etc/Rivera_Thomas_SNIA_100-Year_Archive_Survey_2017.pdf.

[22] Horison Information Strategies. 2015. Tiered Storage Takes Center Stage. http://horison.com/publications/tiered-storage-takes-center-stage/.

[23] Eduard Valera Zorita, Pol Cuscó, and Guillaume Filion. 2015. Starcode: Sequence clustering based on all-pairs search. *Bioinformatics (Oxford, England)* 31 (01 2015). https://doi.org/10.1093/bioinformatics/btv053

[24] Jiannan Wang, G. Li, and Jianhua Feng. 2010. Trie-join. *Proceedings of the VLDB Endowment* 3 (2010), 1219 – 1230.

[25] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of International Conference on Management of Data (SIGMOD)*. 85–96. http://doi.acm.org/10.1145/2213836.2213847

[26] Wei Wang, Jianbin Qin, Chuan Xiao, Xuemin Lin, and Heng Tao Shen. 2013. VChunkJoin: An Efficient Algorithm for Edit Similarity Joins. *IEEE Transactions on Knowledge and Data Engineering* 25, 8 (Aug. 2013), 1916–1929. https://doi.org/10.1109/TKDE.2012.79

[27] Chuan Xiao, Yi Wang, and Xuemin Lin. 2008. Ed-Join: An Efficient Algorithm for Similarity Joins With Edit Distance Constraints. *PVLDB* 1 (08 2008), 933–944. https://doi.org/10.14778/1453856.1453957

[28] Haoyu Zhang and Qin Zhang. 2017. Embedjoin: Efficient edit similarity joins via embeddings. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 585–594.