POLITECNICO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING

MASTER's Degree Thesis

Cross Architecture Edit Similarity Join for DNA Data Storage Using oneAPI

Supervisors

Prof. PAOLO GARZA

Prof. RAJA APPUSWAMY

Candidate

EUGENIO MARINELLI

APRIL 2021

A mamma e papà.

Abstract

The amount of data gathered by enterprises is expected to increase significantly in next years. The main problem related to the growth of data is represented by the cost-effective archival of such data. This is due to the physical limitation presented by the magnetic media currently used for data storage. An alternative to the contemporary magnetic media is represented by synthetic DNA, still under study, but that presents some interesting properties - in terms of durability and high density - that makes it very promising. However, the process of retrieving data from DNA is limited by the `read consensus` stage whose objective is to identify millions unique strings among hundreds millions of noisy copies. This involves similarity join algorithms that are not able to scale to such dataset because of both the complexity of the metric used - edit distance - and for their single-threaded design.

In this thesis, we present OneJoin, a cross-architecture edit similarity join that can exploit multicore CPUs, integrated GPUs, and multi-vendor discrete GPUs using a single code base. OneJoin is implemented with oneAPI - an open, standards-based unified programming model for achieving portable data parallelism. In this work we present the main aspects of oneAPI and we describe the design choice of our algorithm. Moreover, we present an end-to-end DNA data decoding pipeline based on OneJoin and the experimental evaluation of our algorithm compared with the existing solutions. We will show that OneJoin can achieve up to 21× speed-up over the other state-of-the-art algorithm, reducing the DNA data decoding time from several hours to a few minutes.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the advance of AI and analytics, the amount of data produced and gathered by enterprises saw an enormous increase, reaching the 160 Zetta byte Global Datasphere by 2025 [1]. According analysts, only a small minority (20%) of the total can be defined `hot` data, while the so called `cold` [2, 3] data account for the 80% of the total. We define `cold`, the data accessed rarely. This data are characterized by having the highest growth rate and a longest lifetime with a retention period of 50-60 years [4].

With the current technology, this data are stored in NAND flash, HDD, or tape. The main issues of these technologies are the `density` and the `durability` - characteristics that makes difficult the cost-effective storage and management of cold data.

For this reason, the research in this sector investigated on alternative technology that could improve both these properties. They found the perfect replacement of the current technology in the Deoxyribo Nucleic Acid, a.k.a. `DNA`. The DNA is a macro-molecular made by 4 smaller molecules called nucleotides: `Adenine` (A), `Cytosine` (C), `Guanine` (G), and `Thymine` (T). DNA used for data storage is a single-stranded sequence of these nucleotides, called oligonucleotide (oligo). Figure 1.1 shows the key steps performed in order to use DNA as a storage media. The first is to use an encoder to map the digital data into sequences of oligos. These oligonucleotide sequences are used to synthesize physically, through chemical processes, the DNA one basis at time. Because of physical limitation, the DNA

sequence cannot be longer than a few hundreds nucleotides. This implies storing data by using millions of small sequences.

After the DNA is synthesized, the reverse process starts: retrieving data from the DNA. It starts with the sequencing, a process consisting in reading the original oligos and producing a very high number of copies called `reads`. Because of the physical limit of this procedure, these reads presents some `errors`, such as insertion, deletion and/or modification. However, the noise introduced in this step is not a problem if we use a `consensus` algorithm to infer the original oligos. In the end original digital data are read back starting from the consensus results - the oligos inferred - by using a decoder. Basically consensus consists in infer position-wise the original nucleotides by a majority among all noisy copies. More details about the consensus procedure are given in the Section 3.3.

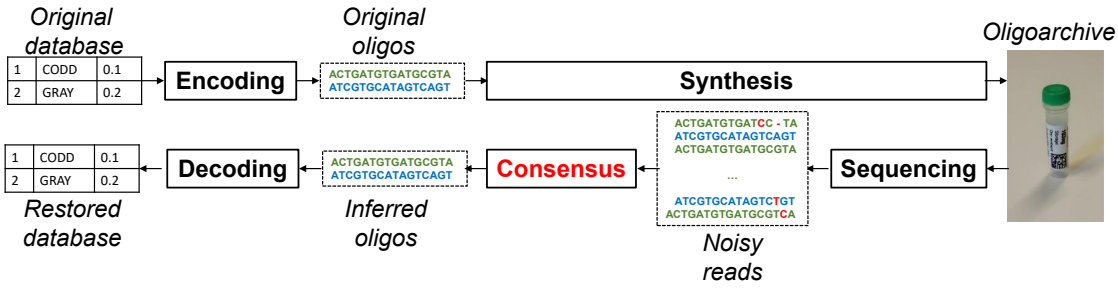


Figure 1.1: DNA data storage pipeline.

Figure 1.2 [5] compares the storage density between different DNA and currently adopted storage media. As we can see, the DNA capacity is estimated being seven orders of magnitude higher than the best expectation for a tape. Also, it has the advantage of a potential longevity of several millennia.

Although the consensus procedure is pretty straightforward if we already know all noisy copies of the original oligos, in practice we need to compute all similar `reads`. This configures consensus as a large scale similarity join problem. Given a collection of strings (reads in our case) we need to find all string pairs that are similar according a certain distance function. This makes the consensus steps the computational bottleneck in the DNA data archival pipeline. Since the errors introduced during the sequencing can be insertion, deletion or substitution, it is necessary to use the `Levenshtein`, a.k.a. edit distance. The main issue related

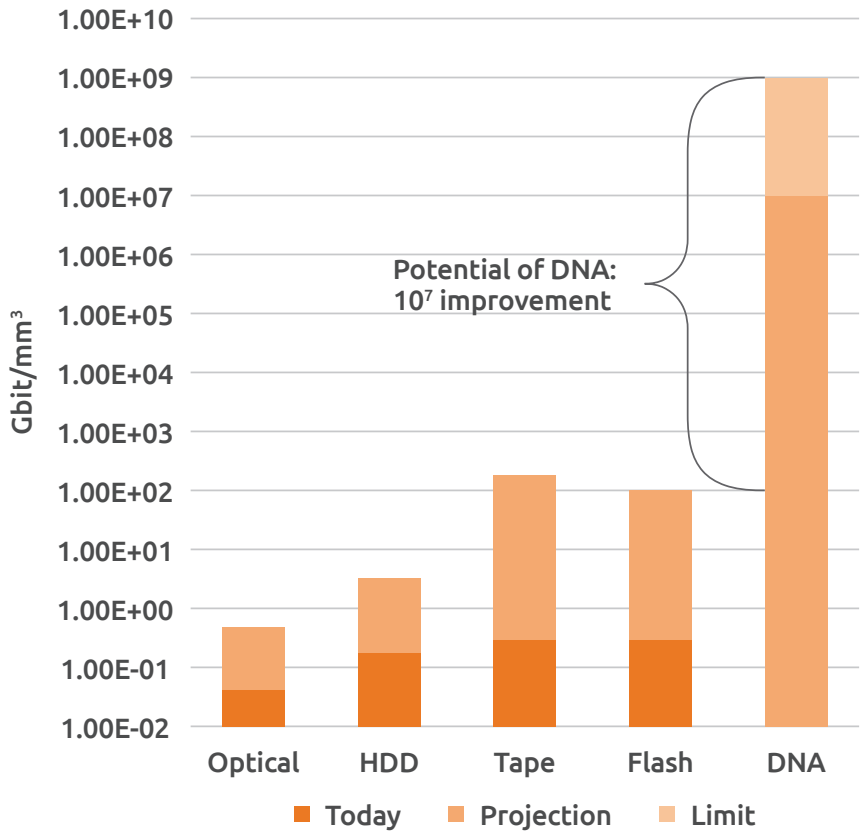


Figure 1.2: Volumetric information density of different storage media.

to this metric is its complexity - since it has no sub-quadratic solution - and this complexity is unacceptable when involves millions of strings.

This metric issue can be addressed by changing the `space` in which strings are represented, trading accuracy for complexity. By using randomized algorithms we can transform the original strings in an embedded representation such that the edit distance is approximable with the Hamming distance. Although it is an approximation, using the Hamming distance makes easier the problem.

EmbedJoin is a strings similarity join algorithm that exploits the edit-to-Hamming embedding together with the Locality Sensitive Hashing to filter pairs that are not similar for sure by limiting the exact edit distance computation to the only pairs that pass thorough the selection phase. Despite the approximation of the exact similarity join algorithm, EmbedJoin showed to have high accuracy also

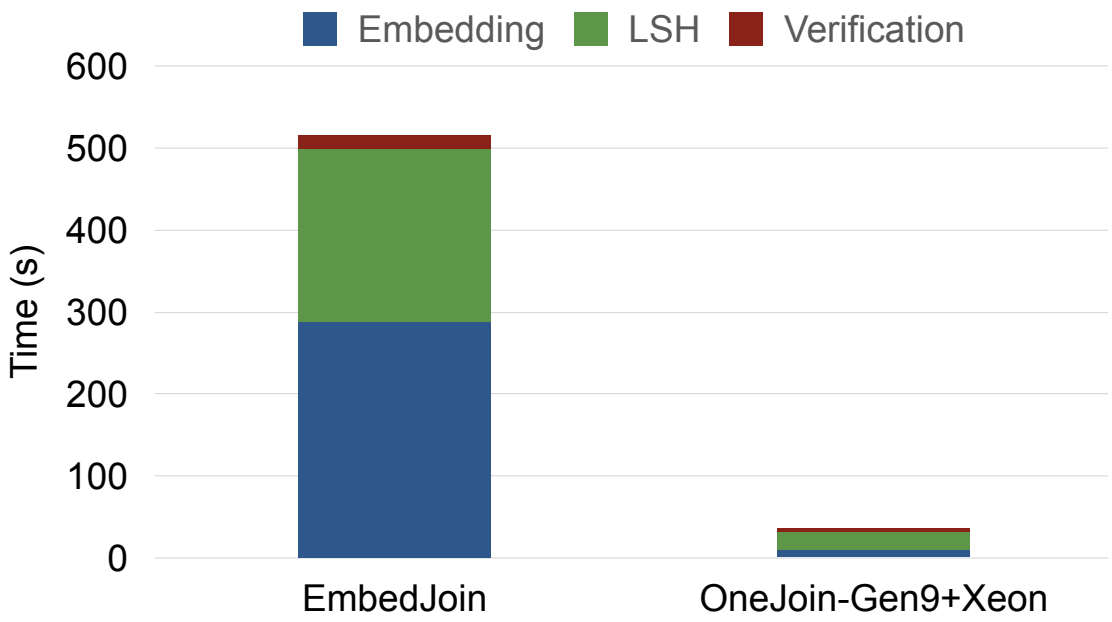


Figure 1.3: Execution time break down of EmbedJoin vs OneJoin for computing edit similarity over a dataset of 450k strings.

for high edit distance thresholds and outperforms the state-of-the-art. Specifically, the paper [6] shows that in all their experiments the algorithm can find above 90% of all similar strings found by the exact algorithm; moreover, in the majority of the cases the figure reaches the 99% of all pairs. This makes this algorithm the perfect candidate to read clustering in DNA storage. However, it still has limitations: it cannot scale over the millions of strings characterizing the DNA storage problem because of its single-threaded nature that limits the execution on one CPU only.

Figure 1.3 shows the execution time of EmbedJoin under a dataset of 470000 strings of length 5000. We will show details about the experiment in the Chapter 4; however, it is important to do two observations that will motivate all the following study. The first point is that the execution time is about 10 minutes for 470000 strings, that is far from the millions strings we have to deal with in the DNA use case. The second is that EmbedJoin minimize effectively the edit distance impact in the total time: the latter contributes only for the 5%, while the most of time is spent in the filtering phase made by *embedding* and *LSH* that is - for design choice - sequential. Thus, while for small dataset this algorithm is best choice, it is

unsuitable to deal with the DNA `reads`.

The major contributions of this thesis are the following.

- We introduce OneJoin - the first cross-architecture, database join operator implemented using oneAPI - an open, standards-based unified programming model for achieving portable data parallelism. We shows various design aspects that led to transform the EmbedJoin, single-threaded CPU-based algorithm, into OneJoin, oneAPI-based data parallel algorithm that is able to execute on CPU, integrated GPU and PCIe-attached discrete GPU. A first comparison between EmbedJoin and OneJoin is shown in Figure 1.3. The OneJoin algorithm, run by using the 6-cores of Xeon CPU, demonstrate being 15× faster than EmbedJoin, providing a 30× speed-up in embedding time, a 10× speed-up in LSH phase and a 6× speed-up in verification time, by processing 470000 strings in less than half a minute.

- We present an efficient algorithm to address the DNA consensus problem, that overcome the bottleneck of identifying similar reads by using OneJoin, solving the consensus problem in few minutes on a commodity server.

- Using both a real-world experimental and synthetic data, we evaluate OneJoin algorithm to address the following points: (i) analyze design trade offs in using DPC++ and oneAPI for developing OneJoin, (ii) compare the performance and scalability of OneJoin with state-of-the-art similarity joins, and the OneJoin-based read consensus solution with other alternative solutions, (iii) analyze OneJoin performances under several processor architectures.

The rest of the thesis is organized as follows. Chapter 2 illustrates both algorithmic and systems background relevant to this work. Chapter 3 details the design choices and the implementation of OneJoin; moreover it is presented the end-to-end DNA decoding pipeline based on OneJoin. Chapter 4, shows the experimental evaluation. Finally, we conclude with a discussion of future work in Chapter 5.

Chapter 2

Background

This chapter is conceptually divided into two parts. In the first part, we present the main algorithmic tools EmbedJoin uses to perform the edit similarity join, i.e. CGK Embedding and Locality-Sensitive Hashing for Hamming distance, and the EmbedJoin algorithm itself. In the second part, we show the system background represented by OneAPI and DPC++, highlighting the salient features in order to better understand the following design choices.

2.1 CGK Embedding

The edit distance between two strings x and y is defined as the minimum number of edit operation i.e. insertions, deletions and substitutions, necessary to transform x in y. On the other hand, Hamming distance takes in account only the number of mismatch between the two strings or in other words the number of substitution to transform x in y.

For example, given the two strings $ACACT$ and $GACAC$, their Hamming distance is 5 since there are no matches, but edit distance is 2 since it suffices to add G and remove T

From these definitions, we can make two observations: Edit Distance keep in account also information about the ordering of characters and capture the best alignment between the two strings. However, the Hamming distance has complexity that is linear with the string length, while the edit distance has complexity that

is in the best case quadratic.Given the complexity of this metric, we can rely on metric embedding technique.

Metric embedding techniques are methods consisting in mapping a complex `metric space` into a simpler one. One of these techniques, widely adopted to handle the edit distance, is `CGK-embedding` algorithm, recently proposed by Chakraborthy et al.[7]. It is a randomized algorithm that maps a string x in the edit distance space into a string x' in the Hamming space.

Given two strings x, y of length N taken from an alphabet $\sum$ such that $d_E(x, y)$ - the edit distance between x and y - is less than K and an embedding function f: $\sum^N \rightarrow \sum^{3N}$, then CGK-embedding assures that with probability at least 0.99 the related Hamming distance $d_H(f(x), f(y))$ is bounded by K^2 when $d_E(x, y) < K$. This implies that the distortion D, defined as the ratio $D(x, y) = \frac{d_H(f(x),f(y))}{d_E(x,y)}$, is at most K. Notice that all pair of strings having $d_H(f(x), f(y)) < K^2$ includes all strings having $d_E(x, y) < K$. Thus, as long as the distortion is small, the number of false positive is small. This allows to reduce the time for the exact edit distance verification [6].

Algorithm 1 CGK-embedding

Input: A string $S \in \{A, C, G, T\}^N$, a random string $R \in \{0, 1\}^{3N}$ and a char for padding $P = 0$
Output: The embedded string $S^{'} \in \sum^{3N}$

```
1: i ← 0
2: for j = 0 → 3N − 1 do
3:     if i < N then
4:         S'_j ← S_i
5:     else
6:         S'_j ← P
7:     end if
8:     i ← i + R_j
9: end for
10: return S'
```

The pseudo-code of embedding algorithm is shown in Algorithm 1. In this case the procedure is applied to all strings of length N made by only the character A, C, G, T, representing the DNA reads.

The algorithm - given an input string - builds the corresponding embedded

representation appending to the output string (initially empty) one character at time taken from the input string. The character appended can be the repetition of previous character or the next character in the input string according to the value of a binary random string. In other words, the pointer of the current character in the input string increases or remains the same depending on the random string value, that can be 0 or 1. When the pointer to the input string goes out of the string length, the embedded string is padded with a special character P. In general, P can be any character that is not included in the set of characters making our input dataset S. For sake of simplicity, in Algorithm 1 we use 0 for padding.

What we get is an output string made by input characters that can appear unique or repeated.

Table 2.1 shows the embedding procedure applied to a simple string S, for a given random bit-string R. It shows all values for the two indexes i and j. We report also the corresponding value for R and S according to the related i and j values. Thus, when $j = 3$ and $i = 2$, $S[i]$ is the character to append to the embedded string - C in our example; while $R[j]$ is the value to sum to i, and since it is 0, then i remain the same. Consequently, the next char to append in the next iteration will be the same just appended, i.e. C.

2.2 LSH for Hamming Distance

One of the main advantages of moving from edit distance space to the Hamming space is the having available some useful algorithms that are valid in the Hamming space only and instead not applicable to edit distance. One of the algorithmic tools is the Locality Sensitive Hashing, or `LSH`. EmbedJoin uses this algorithm to identify similar strings without analyse all possible pairs. We provide an overview of LSH [8] here.

We call a family $\mathcal{H}$ of functions (d_1, d_2, p_1, p_2)-*sensitive* for a distance function D if for any p, $q \in U$ (where U is the item universe):

- if $D(p, q) \leq d_1$ then $\mathbb{P}[h(p) = h(q)] \geq p_1$, that is, if p and q are close, the probability of a hash collision is high;
- if $D(p, q) \geq d_2$ then $\mathbb{P}[h(p) = h(q)] \leq p_2$, that is, if p and q are far, the

Input string $S :=$ AGCTCAA

Random string $R :=$ 011001110111001

Iteration	j	$R[j]$	i	$S[i]$	Embedded String
0	–	–	0	A	A
1	0	0	0	A	AA
2	1	1	1	G	AAG
3	2	1	2	C	AAGC
4	3	0	2	C	AAGCC
5	4	0	2	C	AAGCCC
6	5	1	3	T	AAGCCCT
7	6	1	4	C	AAGCCCTC
8	7	1	5	A	AAGCCCTCA
9	8	0	5	A	AAGCCCTCAA
10	9	1	6	A	AAGCCCTCAAA
11	10	1	7	0	AAGCCCTCAAA0
12	11	1	8	0	AAGCCCTCAAA00
13	12	0	9	0	AAGCCCTCAAA000
14	13	0	10	0	AAGCCCTCAAA0000
15	14	1	11	0	AAGCCCTCAAA00000

Table 2.1: Example of embedding procedure.

probability of a hash collision is low;

where $h \in_r H$ are hash functions randomly sampled from the family of hash functions $\mathcal{H}$

Let's given two bit-string p and q of length N. In the Hamming distance case the hash function is defined as the i^{th} bit of these strings. Thus, if their Hamming distance is $d_H(p, q)$, i.e. the number of bit-wise different bits in the two strings, then the probability of taking a string bit in a random position that is the same in both strings (i.e. there is a collision) is $1 - \frac{d_H(p,q)}{N}$. At this point we can define the bit-sampling LSH family for Hamming distance as:

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

that is $\left(d_1, d_2, 1 - \frac{d_1}{N}, 1 - \frac{d_2}{N}\right)$-sensitive for the two Hamming distances $d_1 < d_2$. Notice that, playing with probabilities, the algorithm introduces both false positive and false negative, that means dissimilar string are grouped together in the same

hash bucket while similar strings are not identified because they go in different hash buckets. One way to reduce the false positive and negative rate is using the AND-OR construction. The AND construction consists in concatenating m bit-sampling hash functions, and using these bit to put a string in a bucket. The OR construction consists in repeating the bucketization of multiple m-bit groups, such that if two string end up in the same bucket for at least a certain number of these hash groups, then they can be considered being similar.

To be more formal, we define the AND construction as

$$f = h_1 \circ h_2 \circ ... \circ h_m, where \forall i \in [m], h_i \in_r H \tag{2.1}$$

such that for $x \in U$, $f(x) = (h_1(x)h_2(x)...h_m(x))$ is a vector of m bits.

Then multiple $f(x)$ are grouped together as

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

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 $f_j(x) = f_j(y)$. It has been shown that g(x) is $(d_1, d_2, 1-(1-(d_1/N)^m)^z, 1-(1-(d_2/N)^m)^z$-sensitive. By playing with m and z we can increase the gap between the two probabilities and reduce the misclassification rates.

2.3 EmbedJoin

EmbedJoin is a similarity join algorithm based on edit distance. The purpose of the algorithm is to find all pairs whose edit distance is lower than a given threshold. EmbedJoin uses both the algorithmic tools defined above, i.e. `CGK-embedding` and `LSH` in order to limit the search in the space of all possible pairs. The pseudocode in Algorithm 2 describes a simplified version of algorithm. We can identify 4 main phases: *initialization*, *CGK-embedding*, *filtering*, and *verification*.

At the beginning, we have a set of input strings, and some parameters: the threshold K for the edit distance, the number of hash functions z, each of them made up of m bits.

In the `initialization phase`, the input dataset is sorted at first according to the strings length and then alphabetically (lines 1-2). Afterwards, all hash functions are initialized. They are stored in form of a matrix D made by random numbers.

Each matrix cell corresponds to a bit in the embedded string. In particular, each column correspond to the m bit-sampling hash function h_i of the AND-construction, while the rows represent the set of all the f_j function of the OR-construction (lines 3-4). At this point, the `CGK-embedding phase` can start. In this phase (line 6), it is applied the previously described Algorithm 1. Having mapped all input strings in the Hamming space by means of CGK-embedding phase, we enter in the filtering stage. The `filtering phase` is designed to process sequentially one string at time. Each input string s_i is put in a hash bucket (line 9). In particular, the f_j function is applied to the embedded version t_i of the strings s_i. This is repeated z times, i.e. once for each hash function f_j defined during the initialization (line 8). When a string s_i is put in a hash bucket according the hash functions (line 9), it is compared against all strings s_l already present in that bucket (line 10). At this point two operations are performed: (1) the length of the two original strings are compared; (2) the hash bits of the two embedded strings are compared to check if they actually are equal (line 14). The first condition is used to avoid unnecessary further comparisons. Indeed, if the difference of lengths is greater than the edit threshold K, then it is impossible that the edit distance is lower than K. Moreover, being all strings in the bucket `sorted by length`, we can remove s_l from the bucket because all next s_i will have for sure a length difference greater than K. The second condition is to manage the "collisions". Recall that f_j is made up of m bit-sampling hash function in AND construction. This correspond to the m chars in the embedded string. However, we can obtain the same hash buckets, also if chars are different. However, in order to consider two strings as candidates, all bits of $f_j(t)$ must be equal. If the second condition is met, that pair is added to the candidate set (line 15). Finally, given the OR-construction, the same pairs can appear in multiple buckets. Thus, a duplicate reduction is performed.

At last stage - `verification phase` - each candidate pair is verified as having distance lower than K by computing the exact edit distance (lines 21-25).

Two observation are now necessary. Using the CGK-embedding and LSH reduces drastically the running time. This make EmbedJoin outperforming all other existing algorithms. However, the optimization in the buckets updating and buckets searching operations takes the 90% of the running time (see Figure 1.3) and while it proves to be efficient for small dataset having strings of different lengths it

is certainly inefficient to scale over hundreds of millions to billions of strings having all same length as in the DNA reads clustering problem.

Algorithm 2 EmbedJoin

Input: A set of strings $\mathcal{S} = \{s_1, s_2, ...s_n\}$, distance threshold K, parameters z, m
Output: $\mathcal{O} \leftarrow (s_i, s_j) \mid s_i, s_j \in \mathcal{S}; i \neq j; ED(s_i, s_j) \leq K$

$\mathcal{C} \leftarrow \emptyset$
Sort $\mathcal{S}$ by string length and then by alphabetical order
for each $j \in [z]$ **do**
    Initialize hash table $\mathcal{D}_j$ by generating a random hash function $f_j \in \mathcal{F}(m)$, where $\mathcal{F}(m)$ is the set of Hamming hash functions defined by the AND construction.
end for
T $\leftarrow$ CGK-Embedding($\mathcal{S}, z, m, r$), where T is the embedded dataset
for each $s_i \in S(sorted)$ **do**
    for each $j \in [z]$ **do**
        Store s_i in the $f_j(t_i)$-th bucket of D_j
        for each string s_l, $i \neq l$ already in the $f_j(t_i)$-th bucket of table D_j **do**
            if $\mid s_i \mid - \mid s_l \mid \geq K$ **then**
                Remove s_l from D_j
            else
                if $f_j(t_i) = f_j(t_l)$ $\forall$ bits of f_j **then**
                $\mathcal{C} \leftarrow \mathcal{C} \cup (s_l, s_i)$
                end if
            end if
        end for
    end for
    Remove duplicate pairs in $\mathcal{C}$
    for each $x, y \in \mathcal{C}$ **do**
        if $ED(x, y) \leq K$ **then**
            $\mathcal{O} \leftarrow \mathcal{O} \cup (x, y)$
        end if
    end for
end for

2.4 oneAPI and Data-Parallel C++

Modern computing systems are characterised by being heterogeneous, presenting a mix of `scalar`, `vector`, `matrix` and `spatial` architectures deployed on multiple

types of devices such as `CPU`, `GPU`, `FPGA` and `AI` accelerators. One of the objective of this thesis is to redesign the main steps of EmbedJoin in order to make it able to run on multiple hardware accelerators. In particular, we will focus on the parallel execution on CPU and GPU. The lack of a common programming languages and APIs, as well as supporting tools and the presence of proprietary solutions led developers to have to adapt their software to different contexts.

OneAPI is an open, standards-based unified programming model that overcomes these problems making the software development independent from the accelerator architecture. The main component of oneAPI is the programming language `Data Parallel C++`, a.k.a. `DPC++`, an open-source implementation of SYCL standard. SYCL is an industry standardization effort with the objective of extending C++ programming language with the support for a cross-platform data parallelism. DPC++ relies on Clang and LLVM compilers and implements all data-parallel constructs defined in the SYCL standard with some Intel specific extension. In this chapter we provide an overview of the main DPC++ features. For further details on oneAPI, please refers to the Data Parallel C++ handbook [9].

Listing 2.1 shows a simple DPC++ example, that highlights the key aspects of the programming language that are relevant for OneJoin design. As it is a C++ extension, we can basically use all standard C++ features, to define variables, data-structures, etc. However, DPC++ introduces some other constructs unknown to the standard C++. The first observation is that as all other accelerators programming model, also DPC++ distinguish between a `host` and a `device`. The host usually is the CPU, although the standard does not put any limit about it except that the host has to support the full C++17; while the device can be the CPU, GPU and/or other hardware accelerators. Notice that CPU can be both host and device. The host is in charge of commanding all devices. It is responsible of submitting data-parallel tasks, gathering results from devices and synchronize jobs between different devices. The host commands devices by means of `queues`. Each queues is associated to one device and although each device can be controlled by multiple queues, one queue can be associated with only one device. A queue contains all tasks the host submits to the device and that are ready to be executed. The device is connected to a queue by means of a `device selector`, that is passed as constructor parameter of the queue object. Both queue and device

selector are shown at line 4. Devices run `kernels`. In general a kernel is the code - implementing a data-parallel task - that is executed on a certain device. In DPC++ there are several constructs that implement different forms of kernels, with their own syntax and execution model. In particular, we present two types of kernels, namely `Basic Data-Parallel Kernel` and `Explicit ND-Range Kernel`.

Basic data-parallel kernel is expressed with the `parallel_for` function, as shown at lines 14, 23, 32. This kernel is characterized by an execution space called *range* and the function to be executed. The execution space defines the number of the kernel instances - also called `work-item` - and it can be 1, 2 or 3 dimensional. A work-item can be seen as a thread executed in parallel with other threads. During the kernel execution a single work-item is identified by an `id` object - a sort of coordinate in the execution space. The dimensionality of the kernel can be chosen in such a way that it matches the dimensionality of the data.

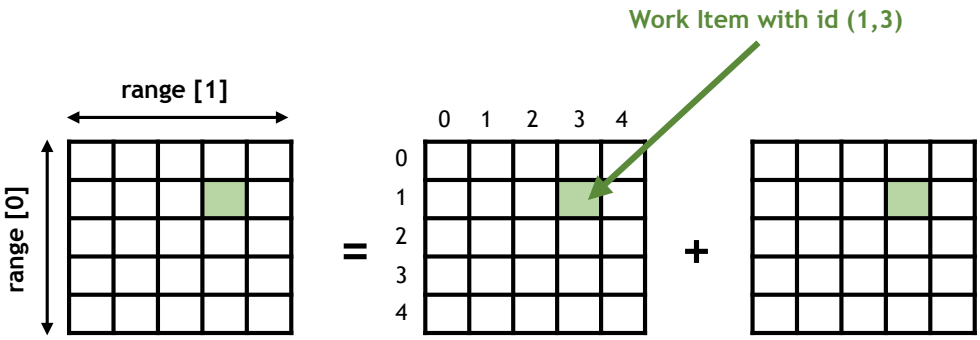


Figure 2.1: Execution space for a 2D-kernel implementing matrix addition.

Figure 2.1 shows the execution space for a 2D basic kernel implementing a matrix addition. Let's assume we want to perform a simple parallel addition between two 5×5 matrix. This operation can be decomposed in 25 sum operations between all element in the two matrix. As these operations are independent, each of them can be easily computed by a different thread. This can be done designing a parallel_for

having a 2-dimensional range whose dimensions match the matrix dimensions. This led to create 25 work-items - organized according to the cells of a 5×5 matrix - such that given a thread identified by the id(i,j), that thread takes the element in position (i,j) in the two input matrix and assign that sum to the (i,j) position in the output matrix. This kernel form does not guarantee any synchronization between work-items, and the execution of work-items can happens in any order. Moreover does not provide to developers any support to force the execution order or any form of synchronization. That makes this kernel simple and suitable for problems that are *embarrassingly parallel*, i.e. in which there is no relation between operations carried out by different kernel instances. If we want to force synchronization at work-item level, we need to rely on more advanced kernel forms.

The explicit ND-Range kernel is a more advanced kernel form, that provides more control over the memory accesses and synchronization among work-items. But just because it is a more advanced kernel, it does not mean that it is suitable for any case. In general it is useful when we want exploit the data locality in our kernel. As for the basic kernel, also this one is implemented with a `parallel_for` construct. However, in this case it requires two n-dimensional execution range. The first one expresses the global execution range - as in the basic kernel case; the second one defines the `local range`. The local range tells us how gather all work-items - defined in the global range - in `work-groups`. A work-group is a set of work-items presenting certain properties. All work-items within a work-group are scheduled concurrently; they can communicate with each other while this is not allowed between work-items in different groups; they can access a their own `local` memory, that in some devices is implemented with a dedicated fast memory; work-items within a groups can be synchronized by means of group barriers, that guarantees that at a certain point in the kernel all work-items are done with all operations preceding the barrier before continuing the execution. Notice that work-group execution can happen in any order.

These concepts are explained better with an example. Figure 2.2 shows how work-item and work-groups are mapped in a matrix multiplication case. In the example, the execution space of the kernel is made by 6×6 global range and a 3×3 local range. Each generic cell represent a work-item, while the red cells represent the work-items belonging to the same work-group. Thus, the kernel will execute

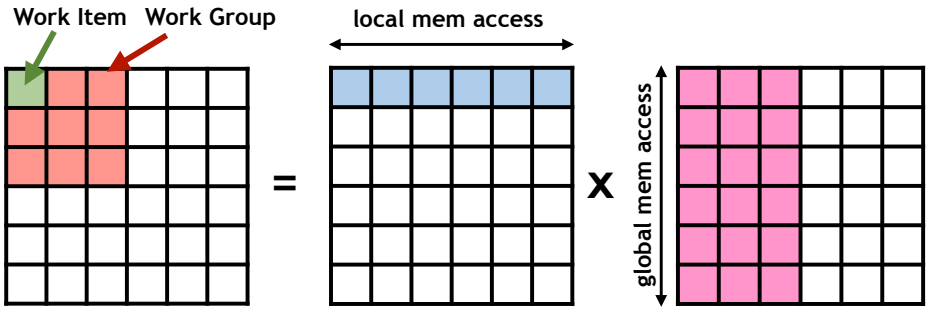


Figure 2.2: Work-items and work-groups mapping for matrix multiplication.

6×6 work-items divided into four groups, each containing 3×3 work-item. As it is an operation involving many times the same data, it can benefits from the use of some caching mechanism. Generally, kernels works with two different memories, a global memory and a local memory. How this memory are actually implemented is device specific. Local memory is faster than global memory, so we can see it as a sort of cache. For this reason we can use the local memory to reduce the latency in accessing data during our operations. Notice that local memory can be accessed explicitly only when we use the advanced kernel form, and each work group has an its own local memory.

Sticking with our previous example, as we know a matrix multiplication is performed considering all element in a row for the first operand and all element in a column for the second operand. Now, if we consider, for example, the first three adjacent elements in the first row of result matrix, each of them is computed considering three times the blue row in the first matrix and the three violet columns in the second matrix. For this reason, if we load the first row in the local memory we reduce significantly the access time for the first operand, since we perform for the first operand one memory access to the global memory and three accesses to the local memory rather than three accesses to the global memory. Notice that local memory is a limited resources, so we cannot store all rows and columns; for

this reason we limit the local memory usage to the first operand only.

OneAPI provides also another kernel form, called hierarchical kernel. It presents the same characteristics provided by the nd-range kernel, but expressed with a different syntax and with a top-down approach against the bottom-up programming style provided by the kernels described above.

The one used in this project is the basic kernel form, since algorithms implemented within each kernel do not require any synchronization or do not present any locality worth exploiting the local memory. However, this does not exclude a further algorithm redesign that takes in account, where possible, these features as future work.

Another key aspect of DPC++ is the memory management. Devices have a their on-chip memory that is different from the host memory. OneAPI and DPC++ provide two way to manage memory: Unified Shared Memory (`USM`) and `buffer` objects (line 8 of Listing 2.1). The USM allows to allocate in C malloc-fashion memory for data, and this allocation can happen in host memory, device memory of in a section of memory shared by both host and devices. It makes easier to integrate existing C++ code with the DPC++. For example, if there are functions accepting pointers as parameters, this function are usable without any change. USM is not used in our project, thus for further detail please refer to the book [9]. Our design exploit `buffers` object for data allocation. `Buffers` provide an abstraction over the memory management, since they represent data object rather than specific memory addresses. One of the advantages of using `buffers` is the automatic management of data movement `host` $\leftrightarrow$ `device`. Data - initially allocated in the host memory as vector of C++ standard library or any other data structure - are moved by the runtime in the device memory in a way completely transparent to the developer. To allocate a buffer and fill it with data already present in the host, it is sufficient to pass to the buffer constructor the starting pointer of data and their size; then data will be automatically copied in the buffer. Buffers only are not sufficient. In order to access data in the buffer we need to pass through an `Accessor` object - lines 13,22,30,31 of Listing 2.1. Accessors are associated to one buffer only and they can have an access mode, such as read, write and read_write. Accessor objects have a key role in tracing data dependency across all kernels. Indeed, thanks to accessors the runtime is able to track all data dependencies by building Directed

Listing 2.1: DPC++ code listing

```
int main() {
    //Create a device queue for GPU
    gpu_selector gselector;
    queue Q(gselector);

    //Create Buffers A and B
    auto R = range<1>{ 16 };
    buffer<int> A{ R }, B{ R };

    //Submit Kernel 1
    Q.submit([&](handler& h) {
        //Accessor for buffer A
        accessor out(A,h,write_only);
        h.parallel_for(R,
            [=](id<1> idx) { out[idx] = idx[0]; }
        );
    });

    //Submit Kernel 2
    Q.submit([&](handler& h) {
        //Accessor for Buffer B
        accessor out(B,h,write_only);
        h.parallel_for(R,
            [=](id<1> idx) { out[idx] = idx[0]; }
        );
    });

    //Submit task 3
    Q.submit([&](handler& h) {
        accessor in (A,h,read_only);
        accessor inout(B,h);
        h.parallel_for(R,
            [=](id<1> idx) {inout[idx] *= in[idx]; }
        );
    });

    // Get result back to host
    host_accessor result(B,read_only);
    for (int i=0; i<16; ++i)
        std::cout << result[i] << "\n";
    return 0;
}
```

Acyclic Graph a.k.a. the *task graph* and force the execution of all submitted tasks in the right order. That is a first form of synchronization between kernels, but it is not always sufficient. In some case we need to coordinate the execution between host and devices as well.

In DPC++ jobs submissions is asynchronous, so the control is returned immediately to the host that continues the execution regardless of the jobs submitted. How can we force the host to wait after the submission? A smooth way is provided by the buffer class. As well as the buffer constructor acquires data during the buffer initialization, so the buffer destructor is in charge of moving back data from the buffer to the original host memory address specified in constructor phase. The destructor is called automatically when the host goes out of the scope in which the object is declared. However, it is implemented in such a way that the data movement does not start till all jobs using that buffer are done. This makes the host stuck in the object destructor. Notice that this form of synchronization is possible only if we need to retrieve data from devices. If we do not need to destructs the object, but only wait for the jobs, we can simply call the wait() function, that forces the host to wait for all jobs submitted till that point and the data remain on the device.

The DPC++ program is compiled with the `dpcpp` compiler and the binary generated is called `fat binary` since it contains all the compiled and intermediate code to run on heterogeneous machine. Unlike other data-parallel programming languages, in DPC++ the host and devices code can coexist in the same file. This enables the single-source compilation that allows the compiler to optimize also the devices code across the entire program without distinguish between host code and kernels.

Overall using DPC++ to express data-paralleling computation has the main advantage of implementing an open standard and providing an high level programming model in contrast with other data parallel programming languages that are proprietary such as CUDA or low level such as OpenCL. Moreover, it increase the portability of applications over multiple architectures or over devices of different vendors. Indeed it is enough to change device-selector to switch the execution of the same code on CPU rather than GPU or FPGA, or change the backend compiler to make the code runnable on CUDA devices.

Chapter 3

Design and Implementation

OneJoin is a redesign of EmbedJoin - the sequential algorithm. Conceptually, it implements the same high-level structure as EmbedJoin, consisting in embedding, LSH, and verification. However, OneJoin is made as a collection of data-parallel steps, that makes the algorithm able to exploit the heterogeneous parallelism characterizing the modern servers. The data-parallel steps differ each other both for the parallelism granularity and computational resources required. Given the heterogeneity on algorithmic, workload, and hardware front that OneJoin aims to achieve, it is implemented completely by using DPC++. The chapter is organized as follows. At first we describe the different data-parallel stages. Then, we present the runtime cost estimation and cross-architecture fork–join execution model used by OneJoin to divide the workload such that it can exploit CPUs and GPUs. In the end, we describe in detail how OneJoin implements the end-to-end DNA data decoding pipeline.

3.1 Data-Parallel Edit Similarity with OneJoin

Data-parallel Embedding.

The first data-parallel stage is the Embedding stage. For sake of simplicity we presented in Algorithm 2 a simplified version of the CGK-embedding algorithm, in which each input string is embedded once according to one random bit-string only. However, it can happen that a random bit-string fails to embed with a low

distortion. It has been proven that one way to reduce the impact of this problem is to embed each string multiple times. The paper [6] showed that increasing the number of random bit-strings effectively reduce the distortion rate, and a good distortion implies a lower number of false positive and false negative. We refers to r as the number of random bit-strings.

The embedding stage is implemented as basic data-parallel kernel using the `parallel_for` function. The construct take two parameters: a range and the function that each work-item has to execute. Both these parameters depends on different aspects such as data organization and synchronization level required between work-items. As showed in Algorithm 1, a single embedding procedure is sequential in nature. This implies that if we split this task across multiple threads, it would be required a synchronization among all threads, each one waiting for the previous ones. This would reduce drastically performances. Thus, the solution adopted is to move the parallelism at an higher level, parallelizing rather than the single embedding procedure, the embedding procedure across multiple strings. In other words, given n input strings and r random strings, there will be $r \times n$ threads (or work-items), each of them performing the embedding for one of $\{input\_string, random\_string\}$ combination. This led to design a `parallel_for` having a `range` with two dimensions: the first one giving the index of the input strings and the second one the index of the random bit-string. Thus, conceptually the first r work-item will embed the first string using a different random bit-string. The second set of r work-item will embed the second string, and so on.

Figure 3.1 shows a representation of the execution space of the embedding kernel. Having defined a 2-dimensional kernel, work-items will results arranged in form of matrix whose dimensions are the same of the range. Given a work-item id, we can easily retrieves the parameters we need for the embedding procedure; in Figure 3.1 the work-item having id(1,2) will embed the string 1 by using the random string with index 2. All data structures in OneJoin are stored in a different DPC++ `buffer`. However buffer objects are limited in size and this limitation depends on the device. It could happen that for large dataset, we cannot create a single buffer capable of containing all data. This is especially true for the input dataset and the embedded dataset that are some of the most expensive data structure of the program in terms of space requirement. For this reason we organize the input

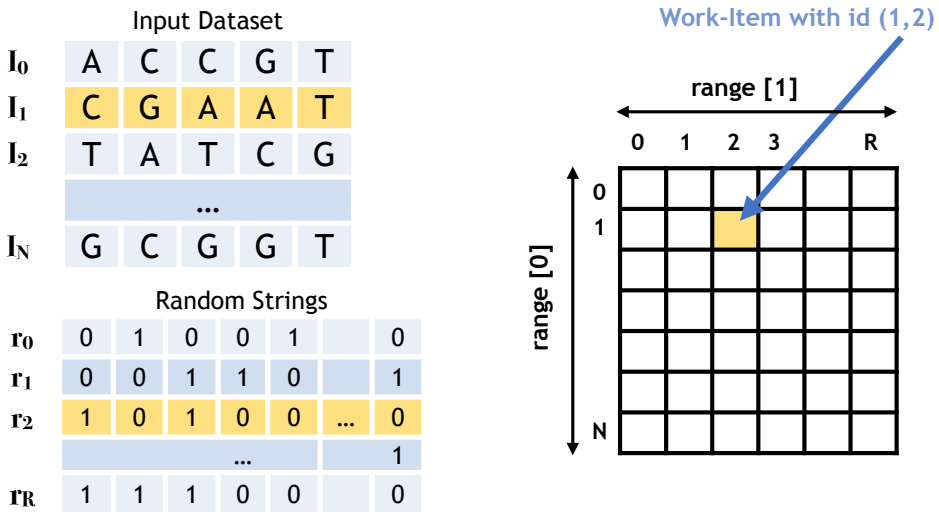


Figure 3.1: Execution space for embedding kernel

strings in batches. The batch size is chosen such that all the embedded strings for a given batch of input strings fit in one buffer. Although the Figure 3.1 shows input strings organized as a matrix of strings, the actual layout is different. Indeed, it can happen that a dataset have strings with various lengths. This makes the matrix organization inconvenient since we should allocate a matrix having rows with same length of the maximum length in the dataset. To overcome this problem we arranged the input strings in a 1-dimensional array. The embedded dataset follows the same structure. Although the access in the array is straightforward within an embedded batch since all embedded strings have same length, in the input strings case, we need an additional vector keeping track of their starting index.

The results of embedding stage is the embedded dataset, that will become the input for the next stage, i.e. LSH. If we think about how LSH works, we can make two observations. The first observation is that there is no need of storing all chars for an embedded string. Indeed, among all the $3N$ characters whose an embedded string is made, the character that will be effectively used are the ones sampled by the m bit-sampling hash functions in `AND` construction. Since there are z of such hash functions, it is enough to keep only $z \times m$ characters, that usually are much

lower than the total $3N$ characters.

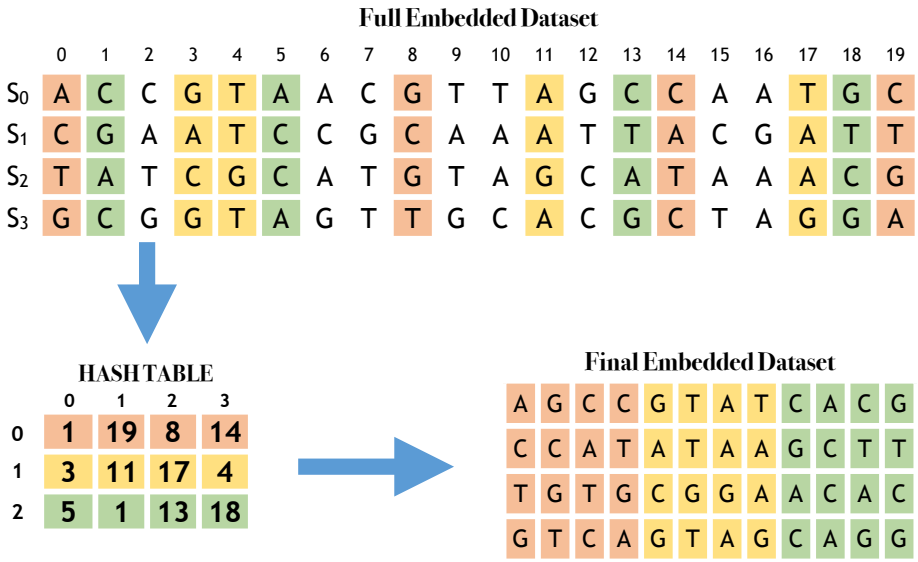


Figure 3.2: Embedded dataset layout

The second observation concerns the ordering of all hash bits selected. This $z \times m$ bits - since represent random positions in the embedded strings - can appear in any order. As we will describe later, each work-item in the LSH stage will be in charge of computing an hash id multiplying all m bits of one hash function only. What could happen is that each thread fetches data in memory locations very distant according the character position. This lead to an increase of cache misses that decrease performances. So what we do in the embedding stage is to reorganize all characters, such that all m bits for one of z hash functions are stored together in sequential positions in the embedded strings. This ensures that in the following stage, a thread can retrieve all data needed in a lower number of memory transactions. The embedded dataset layout is showed in Figure 3.2.

Once all data structures are initialized, the host submits kernels for processing one batch after the other asynchronously. At the end of submission, the host wait to gather all results before continuing, by calling the `wait()` function.

Data-parallel LSH.

Data-parallel LSH implements the sequential LSH filtering stage of EmbedJoin in parallel. It is made internally by two data-parallel kernels, namely bucketization and candidate generation.

Bucketization. The LSH filtering purpose is to hash the embedded strings in hash buckets such that strings similar are grouped together while dissimilar strings end up in different buckets. At this stage, the input data is the embedded dataset produced by the previous embedding stage. The embedded dataset contains r different embeddings for the same input string. Strings are clustered in buckets if embedded strings generated by a given random string produces the same `hash id` for any of z hash functions. The `hash id` for a z-hash functions is computed by considering its m bits. As we know, the m bits represent the character sampled along the entire embedded string. Thus, at first this m character are mapped in a vector u of ASCII-equivalent integers. Then, given a random vector $v \in 0,1..., P-1$, the hash id is computed as $< u \cdot v > modP$, where P is a large prime number and $< u \cdot v >$ is the inner product. This is the same two-level hashing implementation of LSH used by EmbedJoin [6].

The hash id computation is the first data-parallel kernel OneJoin implements in the LSH stage. The kernel is implemented as a 2-dimensional `parallel_for` where each `work-item` computes one hash id using the m bits of one hash function only. Thus, the total number of work-item is $n \times r \times z$, where n is the total number of input strings while r and z the number of random bit-strings used in embedded and the number of hash functions in `OR` construction, respectively. Notice that also in this case a different parallelism granularity are allowed. For example we can decide to assign to a work-item the hash id computation of all the z hash functions or even the computations for all random strings. However, we opted for a finer granularity in order to expose more parallelism. In general, there is always a trade-off between the overhead due to the threads creations and the amount of operations a single thread performs. Finer granularity often implies few operations performed and consequently that the device is not fully utilized because spent the most of time to deal with the threads creation, memory accesses and context switch between a lot of threads. However, a coarser granularity implies a lot of operations

performed sequentially by a thread, that can lead to total computation time closer to the sequential execution. Empirically, in this case we noticed that assigning one hash computation to each thread assure better performances than the other case.

Once all hash ids are computed we need to concretely groups the strings together into hash buckets. The first way to do that is to create $r \times z$ hash tables and inside these hash tables identify the buckets by using the hash ids computed. All this can be easily realized in standard C++ by using a 2-D vector of map objects. However, these are dynamic data structures that are not optimal for devices. A map object do not need to contains all possible values of hash ids and it adds an entry every time a new element is met. This solution is not suitable given the data-parallel design of OneJoin. The problems are mainly two. The first one is that multiple threads require synchronization to update buckets, leading down performances. The second is that - although the number of the hash tables is known and fixed, the number of hash buckets are dynamically determined at runtime according the hash ids. Since in GPU devices dynamic memory allocation is not allowed we cannot add and delete elements in buckets. Thus we should estimate the worst case memory requirement, and allocate in advance the right amount of memory. However,the max memory size to allocate would be given by the max hash id we can have, i.e. P - where P is a large prime number - for each hash table. This is unfeasible for many computing systems. The second way is the solutions we preferred. It consists in using a 1D vector of four-tuples $< t, k, id, i >$, where $t \in [0, r]$ is the index identifying the random bitstring, $k \in [0, z]$ in the index identifying the hash function, the $id \in [0, P-1]$ is the hash id computed in the kernel and $i \in [0, N]$ is the index of a string in the initial dataset. However in this case we do not need to pre-allocate the entire range P for all r random strings and z hash functions. Each tuple is assigned to one work-item for the hash id computation, and the vector is sorted in the host such that all element having the same tuple $< t, k, id >$ - values identifying the hash buckets - are adjacent in the vector. Notice that the vector size is fixed and automatically determined.

Disadvantage of this solution is not knowing where a bucket start and where it ends. After the kernel execution, the entire vector is scanned and all possible pairs are added in a vector. This procedure is easy to parallelize as well, however we noticed in all our experiments that also sequentially this sub-step take less

than 1 second and that is more efficient to perform it sequentially than paying the overhead of buffers creation kernel launch.

Candidate generation. The main issue related to the hash id is the fact that multiple strings can generate the same hash id also if characters at the m_i^{th} position are different. Thus, in order to be sure that two strings ended up in the same buckets because their embedded strings have the same m bits, it is necessary a further validation step. It consists in comparing the two embedded strings identified as potential candidates before marking them as such. We called this validation step `generate candidates` and it is the second data-parallel kernel in the LSH stage. The kernel purpose is to validate all intra-buckets strings pairs, with a bit-wise comparison. The actual kernel is preceded by a small initialization stage and followed by post-processing step. The initialization step purpose is retrieving the list of all candidates pairs to validate and allocate the memory for all data structures the kernel requires. The list of candidates is obtained by means of two operations. Having flattened all buckets entries in a 1-D vector implies losing the direct access to the begin and the end of all buckets. Therefore, at first we compute the boundaries for each bucket, and store the starting index and the size of each buckets into an auxiliary vector. Then, passing through this vector we count the number of strings within each buckets and compute the exact number of candidates as $C = \sum_{b=1}^{B} \frac{n_b \times (n_b - 1)}{2}$, where n_b denotes the number of candidates per bucket. This value is a worst case estimation, since it takes in account all possible combinations of strings for each buckets; however some of them may be filtered out after the bit-wise validation. At this point, we scan one more time the buckets vector to get the string indexes for each pair, making them directly available during the kernel execution.

Finally, we compare bits for all pairs using a data-parallel kernel. The kernel takes as input two embedded strings and compare position-wise the embedded strings considering only the character for one hash function. Remember that embedded strings contains only the character sampled by the hash functions. One limitation of some devices such as GPU is that data structure modifications are not supported. Thus, we cannot add or remove element from a vector during the kernel execution. Thus, we store the candidates list retrieved during initialization in the

output vector, and together with the ids, we hold further information about the validation results. This allows to postpone the filtering in the host, by removing all not valid candidates pairs. We filter out candidates from the output vector based on two criteria. The first one takes in account the length of the original strings [1]. Indeed two strings having length greater than the edit distance threshold K, can generate embedded strings that are hashed in the same buckets. However, if two strings have length difference greater than K automatically have edit distance greater than K since it would require at least K insertion to change the first in the second one. Thus, filtering also based on K helps to further reduce the number of candidates. This step is performed also in EmbedJoin. However, in EmbedJoin, this strategy lead to further improvement. Indeed, since strings are sorted by length, if the condition on the lengths is not met, the original algorithm can remove all the remaining strings in the buckets avoiding in this way further comparison. OneJoin cannot do the same - i.e. removing the strings from the buckets - since we do not have inside the kernels dynamic data structure and we need to performs all comparison anyway. In OneJoin case, filtering by length difference just helps to reduce the candidates number to save verification time. The second criteria takes in account the matching between all bits stored in the two strings composing a candidate pair. In order to be a valid candidates the embedded strings must have the same bits in the same order.

In order to filter based on these two information, during the kernel execution we compute and store them together with the strings ids. We compute the length difference between the two strings and we keep a bit that can assume 0/1 value depending on the outcome of the bitwise comparison between the two embedded strings. Both this information allow to postpone the filtering of candidates - that did not pass the validation - after the kernel execution in the host.

Cross-architecture fork–join execution.

So far we showed the main data-parallel stage composing the OneJoin algorithm, namely embedding, bucketization, and candidate generation. Between each of

[1] We mean by original string the ones in the initial input dataset

these three kernels there are some data-parallel operations such as sorting, reduction, duplicate eliminations responsible of reorganizing the intermediate results of OneJoin. These operations are implemented by using functions provided by the OneAPI library - a.k.a. OneDPL (Data Parallel Library) - and/or OneTBB library. OneDPL provides common functions executable in parallel on any devices while OneTBB provides a set of functions optimized for multi-threading on CPU. An evaluation of performances of using these two libraries will be discussed in Chapter4.

As we mentioned, one of the objectives of OneJoin is to fully exploit the hardware available. This can be done by using the appropriate `device selector`, a DPC++ object associated to any of devices available on the platform. Thanks to DPC++, the three data-parallel stage illustrated above are able to run on different devices by specifying a different device selector. However, multiple devices can be used in the same program as well. OneJoin enable the execution on multiple devices by implementing a cross-architecture fork-join execution model. Fork-join based parallelism consists in having a master thread that forks the execution of a task across multiple computing units. Once tasks are done, all parallel threads are joined in the master. OneJoin implements this model as showed in Figure 3.3. The master thread coincides with the host thread, and we use in this project only two devices: CPU and GPU. In correspondence of a data-parallel stage, input data are split and kernels are lunched simultaneously on all devices available implementing in this way the fork stage. After execution ends, results are gathered by the host and the program continue on the master thread only. This is the join stage. In practice, the model can be realized in DPC++ by exploiting the fact that job submission is asynchronous. This allows the host of not waiting for any jobs before the next submission. Thus, we do not need to explicitly create multiple threads, but just make the master submitting all jobs on all devices and then waiting for all of them.

The two main issue related to this solution is the variability of the execution time which depends on the device characteristics. In order to keep all devices equally busy, we introduced a sampling-based cost estimation model based on which host divides the workload proportionally to device performances.

The cost estimation strategy consists in using a small sample of data to measure

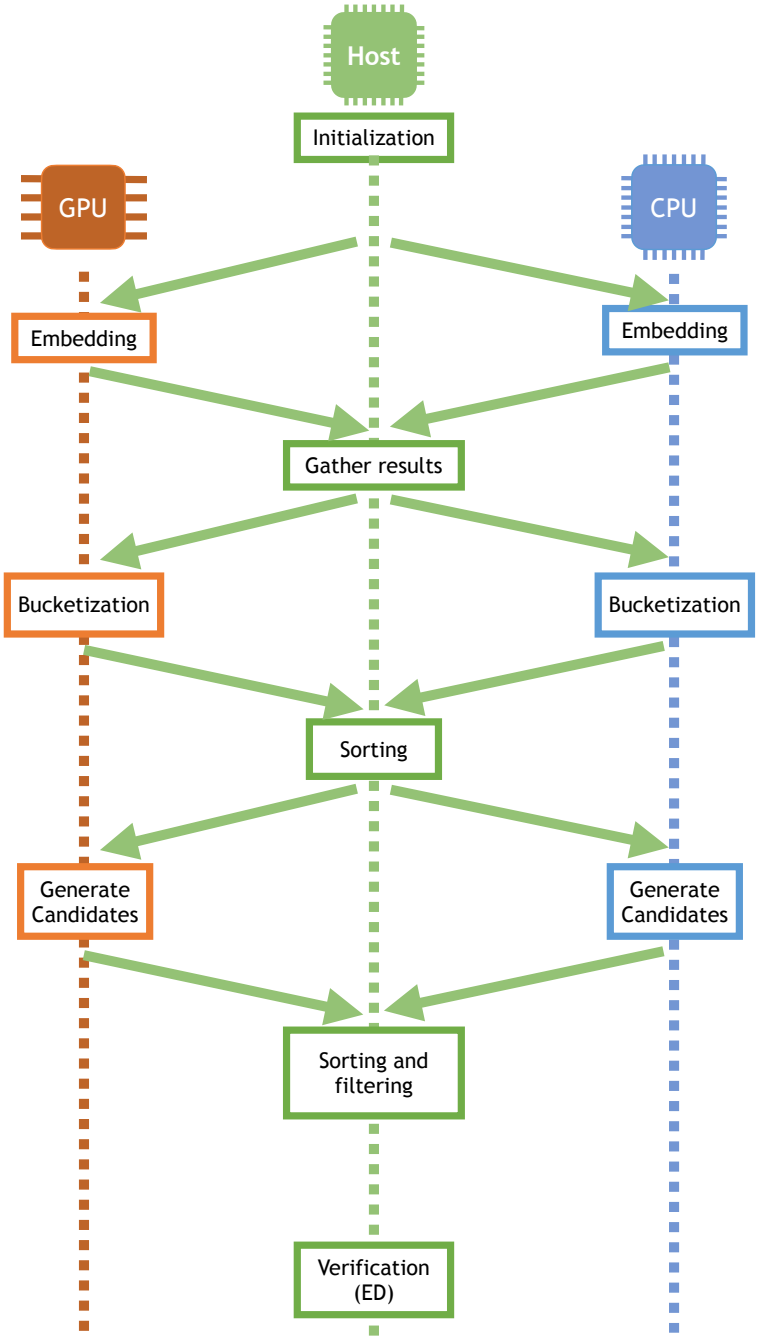


Figure 3.3: Fork-Join execution model applied to our algorithm

execution time on all devices available, and using this estimation to derive the amount of data to assign to each device. Thanks to this simple strategy, kernels on all devices are able to finish their work in roughly the same amount of time. Moreover, the estimation phase is executed sequentially one device after the other; this means that in order gain in performances, it is necessary that the profiling phase is much lower than the actual computation that is executed in parallel.

Edit distance verification.

At the end of LSH stage we obtain a list of candidates among all input pairs. This list is the starting point for the last stage, i.e. the verification. At this point of the program we want to remove all false positive pairs that have passed the filtering stage, on in other words we want to be sure that the candidates in the list have exact edit distance lower than the given threshold. We shows a brief example in which we illustrate the main idea behind the algorithm used to compute it. We can summarize the solution with the following expression.

Given two string S_1 and S_2, where **X** and **Y** are generic chars into S_1 and S_2 respectively; and **a** and **b** the prefixes of the strings S_1 and S_2 till the characters X and Y. Then, we can compute the Levenshtein distance between S1 and S2 as:

$$\text{ED(X,Y)} = \min \begin{cases} ED(a,b) + match(X,Y) \\ ED(aX,b) + 1 \\ ED(a,bY) + 1 \end{cases}$$

We can interpret the expression above in the following way. The first case means that distance between two generic strings aX and bY is equal to the minimum edit distance needed to turn the prefix a into b plus one or zero substitution according to the equality check result between their last character. The second case can be seen as the number of operations we have to perform in order to turn the string a concatenated with X plus one insertion. The last case is symmetric to the second one. This expression can be computed starting from the string with length 0, i.e. the empty string till considering the whole strings. In practice this solution can be implemented as a recursive function that performs the same computation among all prefixes. Specifically, the recursive function call itself three times for different prefixes (by taking off one character); and this is repeated until it reaches the

empty string. Figure 3.4 shows the recursive tree of this function. We can see that each node (except the leaves) has three children nodes, representing the three recursive calls. At this point the complexity is easy to estimate: the three height is $\min(m, n)$, where m and n are the lengths of the two strings (that in this example are the same); each nodes has 3 children nodes. Thus, it recurs $O\left(3^{min(n,m)}\right)$. Notice that the red nodes represent the same sub-problem: edit distance between "AC" and "AG". If we compute the full three we can see that this is not the only case.

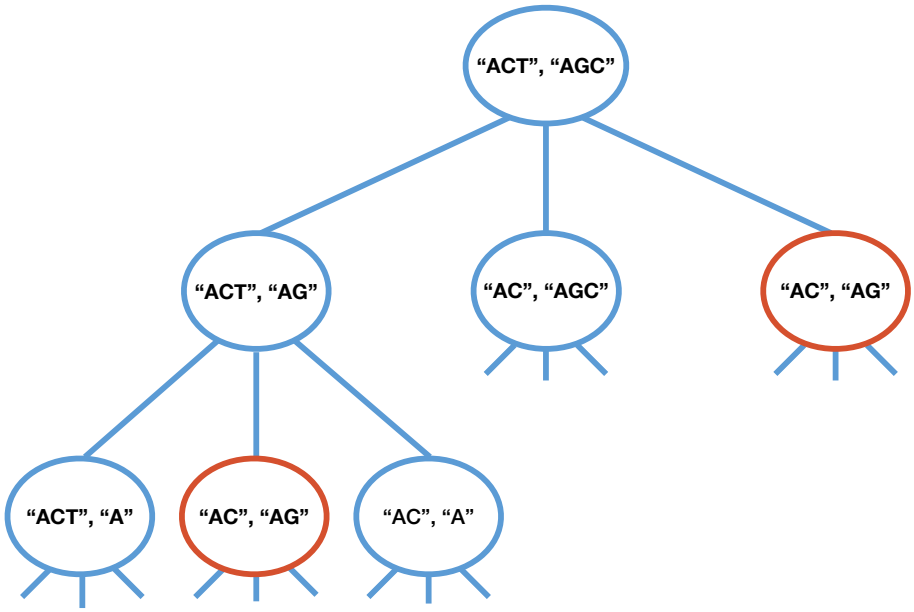


Figure 3.4: Call tree for recursive edit distance function

For this reason the solution is computational expensive, especially if we consider that the edit distance for a certain pair of prefixes is computed many times. In other words, the function try to solve a sub-problem that probably it was already solved in a previous step.

Given the complexity of a recursive algorithm, we can change approach. We can rewrite the solution in form of matrix. We arrange the two strings along the axis of a matrix, and each matrix cell correspond to the last char of the two strings prefixes. At this point we can scan the matrix and fill each cell by using the results computed and stored at previous iterations, basically applying the expression described above.

	“”	A	T	C	G	T	A	C	G
“”	0	1	2	3	4	5	6	7	8
A	1	0	1	2	3	4	5	6	7
T	2	1	0	1	2	3	4	5	6
G	3	2	1	1	1	2	3	4	5
C	4	3	2						
A	5								

Figure 3.5: Example of edit distance computation between two strings

In other words we are trading space for time. The complexity of this solution is polynomial. Indeed we scan a $n \times m$ matrix, so the time complexity is $O(n \times m)$. However, in this case we pay a larger space complexity, due to the allocation of $n \times m$ matrix, while in the recursive solution no auxiliary space was needed.

Figure 3.5 shows a simple example of dynamic programming for computing Levenshtein distance. Each cell of the matrix contains or will contains the edit distance between the prefixes made by the all previous rows and columns. We added also the empty strings as first row and column, since the empty string is a prefix of the strings itself. Notice that we can fill directly the whole first row and first column since the edit distance between the empty string and another string is the length of the string itself.

In the example we consider the cell (4,3). In this case, we are considering the edit distance between the two substrings "AT" and "ATG". The prefix `a` is just "A", while `b` is "AT"; at the same way X is "T" and Y is "G". The values in the considered cell is given by the minimum of the three cells highlighted plus a value. This value is always 1 but in case the minimum comes from the cell in the left corner. In this case we sum 0 if the last character are equal or 1 if the character

are different. The value to add can also be weighted according the importance we give to the substitution,insertion and deletion operations. In this simple case we give the same weight to all the operation, i.e. we always add 1. This algorithmic technique is called Dynamic Programming. It consists of dividing a problem in sub-problems, as well as the recursion technique does, but it stores the solutions to this sub-problems the first time a sub-problem is met. In this way we can build the solution starting from the smaller problems results, without recomputing them when they are equal.

Returning to our algorithm, as in the sequential EmbedJoin, this stage takes a small percentage of the total time (about 5%, as showed by figure 1.3) we parallelise this stage on CPU only, assign one edit distance computation to each thread.

3.2 OneJoin: Full Example

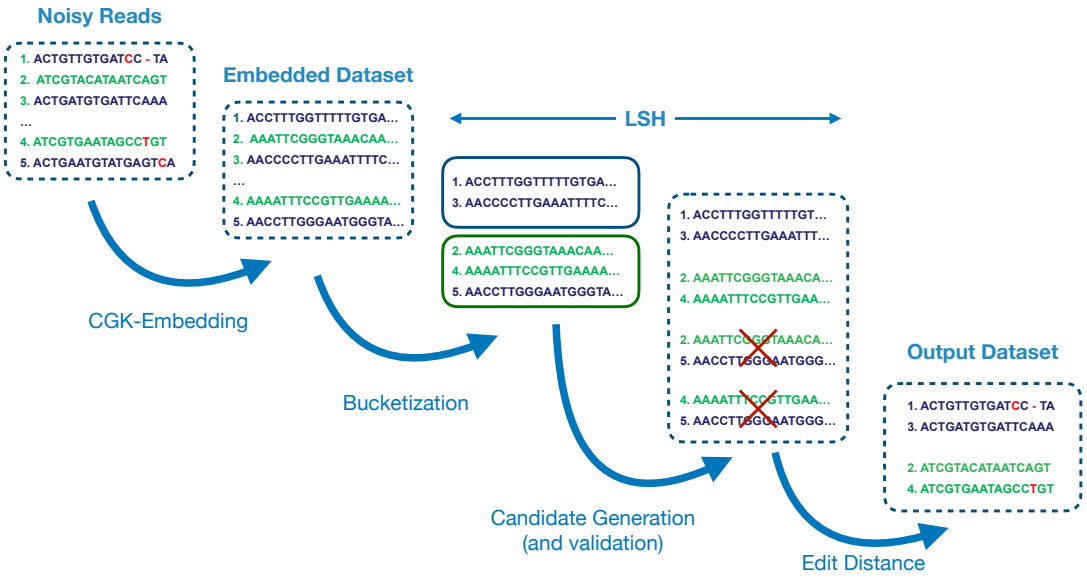


Figure 3.6: Full example of OneJoin main steps.

In this subsection, we present a small example applying to a toy dataset all steps described above. Figure 3.6 shows the main OneJoin stages and the transformations

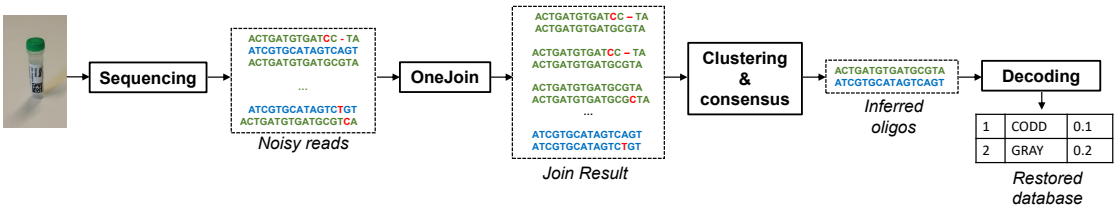


Figure 3.7: End-to-end data restoration pipeline.

applied to the input dataset. The input dataset in made by all noisy reads obtained with the sequencer. The first algorithm step consist in transforming the input dataset in the embedded dataset passing throughout the embedding procedure. We obtain a dataset whose strings have the 3 times the input strings length, and the embedding procedure can be repeated multiple times in order to obtain a good distortion rate. At this point, we are ready to apply the LSH filtering. At first, we separate all embedded strings in buckets; then, going over all buckets, we make all candidate pairs with the strings in each bucket and these candidates are validated with a bit-wise comparison. Notice that indexes in input dataset and embedded dataset are linked, so we can move between the two dataset easily. That is important because the validation of candidates is applied to the embedded dataset, while the verification to the input dataset. The candidate pair who pass the validation is added to the final dataset. The latter is the verified by exact edit distance computation.

3.3 Read Consensus with OneJoin

As mentioned in Chapter 1, the main application of OneJoin is the `read consensus` task. In this section we give more details about it. We start presenting the end-to-end DNA data restoration pipeline showed in Figure 3.7. Notice that except for the consensus stage, all other stages of DNA data storage domain are not detailed, since this is not the purpose of the thesis.

The DNA data restoration pipeline aims to restore DNA previously encoded in synthetic DNA. The main issue related to this process is that the reading procedure of the DNA sequences introduces some errors - insertion, deletion and substitution - that can compromise the decoding of original digital data. In order to overcome

this problem, during the reading step, each oligos is read multiple times, producing multiple copies of the DNA sequence in form of strings called reads; the reads differ each other for some errors in different positions. So, assuming that we can identify what are the similar reads, we can infer the original oligos correctly simply by using a `consensus` procedure. Given a set of reads - supposedly belonging to the same oligos - the `consensus` process consists in inferring the right character of the oligo at each position, by taking the character by majority considering the character in the same position for all reads in the set. It allows in case of errors introduced during the sequencing in some of the reads, that the original oligo can be retrieved anyway. The whole consensus procedure represents the mid stage of the pipeline, and it is made by multiple sub stages. Among these sub-stages, we introduced OneJoin algorithm. In the end, the oligos inferred are used to decodes the original digital data.

The first pipeline stage is called `sequencing`. It takes as input the DNA sequences and generate the corresponding DNA multiple noisy copies, i.e. the reads. The sequencing stage is followed by the consensus process. This stage is made up of multiple sub steps. Indeed, we do not actually know which `read` belongs to the same oligo, and thus we do not have immediately after sequencing the sets of similar reads, but we need to compute them. We can group all strings by means of a `clustering` algorithm that uses the edit distance as similarity metric. So we use the OneJoin algorithm to compute all similar pairs according an edit distance threshold, and then using its result within a clustering algorithm.

We chose for clustering the well-known density-based DBSCAN [10] algorithm. Being a density based clustering algorithm, it works well when clusters are defined by dense regions separated by regions of low density and it can identify clusters of arbitrary shapes. This makes DBSCAN a suitable choice for the DNA reads dataset. However, this does not exclude alternative solutions. In this thesis we do not take in account other clustering algorithms, but we leave it as future work.

The pseudocode of DBSCAN algorithm is showed in Algorithm 3. It consists in scanning the input dataset - the reads in our case - and label each point as noise point or core points, based on the number of points within a ϵ-range from the considered point. The ϵ-range represents the maximum distance within which a minimum number of neighbours for that point are needed in order to consider

Algorithm 3 DBSCAN

Input: Input dataset `DB`; `min_pts`; edit distance ϵ.
Output: The `label` vector.

```
1: C ← 0
2: for each point P ∈ DB do
3:     if label[P] = UNDEFINED then
4:         continue
5:     end if
6:     Neighbours N ← range_query(P,ε)
7:     if Size(N) < min_pts then
8:         label(P) = NOISE
9:         continue
10:    end if
11:    C ← C + 1
12:    label[P] ← C
13:    S ← N \ P
14:    for each point Q ∈ S do
15:        if label[Q] = NOISE then
16:            label[Q] ← C
17:        end if
18:        if label[Q] ≠ UNDEFINED then
19:            continue
20:        end if
21:        label[Q] ← C
22:        Neighbours N ← range_query(P,ε)
23:        if Size(N) ≥ min_pts then
24:            S ← S ∪ N
25:        end if
26:    end for
27: end for
28: return label
```

the point as belonging to a cluster or as noise. In our application the ϵ is the edit distance threshold while the minimum threshold depends on the dataset. When a point is added to a cluster, all its neighbours are analysed and if they are found meeting the min-points condition, they are added to the clusters as well. The bottleneck of the algorithm is the range query. The range query scans the whole input dataset in order to find all points within a distance range for the considered

point, and this is repeated for all neighbours. Moreover, in the DNA reads case, a further complexity is added by the metric used, i.e. the edit distance. Here it is where the OneJoin algorithm comes in.

Before calling the clustering algorithm, we materialize all range query results by using OneJoin algorithm in order to find for each input string, all its similar strings. Then `DBSCAN` access the materialized results directly during the execution. Finally, having executed the DBSCAN algorithm, we can applying the consensus process within each cluster. From consensus we obtain an output dataset containing as many strings as the number of clusters, representing the original oligos.

After clustering, there is the decoding phase. The decoding algorithm converts the quaternary code `ACGT` in the original digital data.

What follows is a toy example showing data at different stages.

Reference Strings
TCCGGAAGTCACAGTTTCAATCCCACTGATCGATGCTCTCTACACCATG
TCGAGACGACCTACGCCGACTCTTGGTAAACGATACGGGGCGATCTATC
ATTCACTAAATTCGGTTAATGAATTCCCCTCGGTACCCTATATTGTACA
AACAAGGAAGCACACGTCCCTTTCGCACAGGAAGCAGTCCAGGCTGGTC
TTTCTTCTACGTGGAACTCAGTATAACGTAGGATAGCGCTGTTGATGTC
CAGTATCGATTTTGCCCAGTGCCATTGCCCCGAAAGAAAATATGCTATT
AACATCAGAGTAATGGTAGGGCTCGGCGACGTAGAATTACTAAACTCGT
TCACCTTTGGTATTCTTACCGGGTAACGCCACCTGTCAAGCTATCCAGC
ATCCACGTACTGTAGTGGAGACCTTACGCCCGAAGTTCGGTGCCAATAT
AAGAAGACTATCGATATCTCCTTAATGGACGGGAACTAAATGTTCACAA

Table 3.1: Reference strings that our algorithm has to infer.

Let's assume that we have 10 strings synthesized as DNA strings representing our encoded digital data, as showed in Table 3.1. These strings are not known in advance and they represent the reference strings that we want to read out. As first step we use the sequencing procedure to read the actual DNA strings. Each of them is read multiple times. However, since the chemical procedure is not perfect, we get from this procedure many noisy copies of the reference strings. We simulate this behavior as showed in Table 3.2. The blue string in Table 3.2 is the reference

Noisy Strings
TCCGGAAGTCACAGTTTCAATCCCACTGATCGATGCTCTCTACACCATG
TCCGGAAGTCAAAGTTTCAATCCCACTGTTCGATGCTCTCTTCACCATG TCCTGAAGTCAAAGTTTCAATCCCACTGATCGATGCTCTCGACACCATG GCCGGAAGTCACAGTTGCAATCCCACTGATCGAGGCTCTCTACACCATG TTCGGAAGTCACAGTATCAATCCCCCTGATCGATGCTCTCTACACCATG TCCGGAAGTCACAGTGCCAATCCAACTGATCGATGCTCTCTACACCATG TCCGGAAGTCACAGTATCAATCCCACTGCTCGATGCTCTCTCCACCATG TCCGGAAGTCACAGTTTCAATCCCACTGATCGATTCTCTCTACACCGTG TCCGGAAGTCGCAGTTTAAATCCCACTGATCGATTCTCTCTACACCATG TCCGGAAGTCAAAGTTTCGATCCCACTGATCGATCCTCTCTACACCATG TCCGGAAGTCACAGCTTGCATCCCACTGATCGATGCTCTCTACACCATG

Table 3.2: Noisy duplication for the first reference strings.

Join output pairs
1. TCCGGAAGTCACAGTGCCAATCCAACTGATCGATGCTCTCTACACCATG 42.TCCGGAAGTCAAAGTTTCAATCCCACTGTTCGATGCTCTCTTCACCATG
1. TCCGGAAGTCACAGTGCCAATCCAACTGATCGATGCTCTCTACACCATG 15.TTCGGAAGTCACAGTATCAATCCCCCTGATCGATGCTCTCTACACCATG
...

Table 3.3: Example of output pairs computed by OneJoin

string, and the black ones are its noisy copies. Indeed we can see that the ten strings are quite identical to our reference, except for some random substitutions of characters. Notice that the reference strings are not within the dataset, and in general at this point all we have are their noisy versions. Table 3.2 shows the noisy duplicate for only the first reference, however this is valid for all reference strings. Moreover, all the noisy copies for all reference strings can appear in any order in

Cluster ID 1
TCCGGAAGTCAAAGTTTCAATCCCACTGTTCGATGCTCTCTTCACCATG TCCTGAAGTCAAAGTTTCAATCCCACTGATCGATGCTCTCGACACCATG GCCGGAAGTCACAGTTGCAATCCCACTGATCGAGGCTCTCTACACCATG TTCGGAAGTCACAGTATCAATCCCCCTGATCGATGCTCTCTACACCATG TCCGGAAGTCACAGTGCCAATCCAACTGATCGATGCTCTCTACACCATG TCCGGAAGTCACAGTATCAATCCCACTGCTCGATGCTCTCTCCACCATG TCCGGAAGTCACAGTTTCAATCCCACTGATCGATTCTCTCTACACCGTG TCCGGAAGTCGCAGTTTAAATCCCACTGATCGATTCTCTCTACACCATG TCCGGAAGTCAAAGTTTCGATCCCACTGATCGATCCTCTCTACACCATG TCCGGAAGTCACAGCTTGCATCCCACTGATCGATGCTCTCTACACCATG
TCCGGAAGTCACAGTTTCAATCCCACTGATCGATGCTCTCTACACCATG

Table 3.4: Example of one cluster produced by DBSCAN and consensus procedure applied within the cluster.

the dataset.

On the resulting noisy dataset, we apply at first the OneJoin algorithm, and results are in the form showed in Table 3.3. Each pair is made by two similar strings and an id that is simply the position within the input dataset[2]. Thus, Table 3.3 says that string in position 1 in the dataset is similar to string 42 and 15. We store this result grouping for each strings, all its similar strings. For example, considering the pairs in Table 3.3, we store a map such as `String 1`: [`String 42`, `String 15`, ...]. This means that `String 1` has `String 42`, `String 15` as its neighbours. However, this is not enough. We need to cluster them, and this is done by applying DBSCAN algorithm.

As DBSCAN needs to compute all neighbours for each point (in our case a point

[2]The id is the index of the array of strings used to store the dataset. It allows to work with integer to represent the pairs rather than strings but also to retrieve easily the strings when needed.

is a string in the input dataset), it can easily take the list of neighbours obtained and stored from the join algorithm. This make clustering extremely fast. The DBSCAN results is a set of clusters; each cluster contains similar strings that are the modified version of the reference string. In other words, we obtain something similar to Table 3.4. The latter shows only one of clusters produced by DBSCAN. However, we expect a number of clusters at least equal to the number of reference strings.

Finally, we can apply the consensus procedure. We consider one cluster at time. So for this example we consider the cluster showed in Table 3.4. We go over each "column" of the set of strings in that cluster, and we compute the frequency of each character. Thus, basically if we consider the first column, there is the "T" that appear 9 times and the "G" once. From this, we deduce that first char of the reference strings is "T". Similarly, if we consider the blue column, we see 7 times the "G", 2 times the "T" and 1 time the "C". The reference string at this position must have "G" as character. As you can notice, we can recover the original string despite the number of errors introduced during the sequencing.

Chapter 4

Evaluation

In this chapter we present a detailed analysis of OneJoin, with the objective to answer to three questions, that are the three contributions of the thesis.

- How does our data-parallel OneJoin perform compared to the other stat-of-the-art similarity joins?

- Is oneAPI able to effectively exploit heterogeneous architecture? How OneJoin performance differs executing on CPU, integrated GPU (iGPU), and a discrete GPU (dGPU)?

- What is the contribute of OneJoin to the DNA storage problem?

The experiments are organized as follows. First, we demonstrate the portability of OneJoin across different processor types, showing the benefit of using oneAPI for the software development. Then, we compare OneJoin with EmbedJoin and other popular join algorithm. Thus, we present a macrobenchmark comparing the join algorithms under several publicly available datasets. Finally, we show the contribution of OneJoin to the end-to-end DNA data decoding problem, comparing it with the state-of-the-art read clustering program.

4.1 Experimental Setup

Hardware Setup. Given the hardware needed for our experiment, we used two servers. The first is Intel DevCloud[1], a cluster of servers equipped with Intel hardware. In particular, the cluster node we use for experiments is equipped with a 6-core Xeon E-2146G CPU clocked at 3.7GHz, 64GB DRAM, and a Gen9 Intel iGPU. The second is 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 DPC++ with O3 optimization. The similarity join algorithm we chose for the comparison are EmbedJoin, AdaptJoin[11], and QChunk[12]. These algorithm have been chosen since they have been demonstrated to be among the best existing algorithms for edit similarity joins [6].

Dataset	n	Avg. Len	Min. Len	Max. Len	$\lvert \sum \rvert$
GEN-20KS	20001	5000	4829	5109	4
TREC	233,435	1217	80	3947	37
UNIREF	400,000	445	200	35213	25
GEN-470KS	470,492	5000	4841	5152	4

Table 4.1: Parameters of datasets used in this work.

Datasets. All benchmarks comparing the OneJoin with other join algorithms use three publicly-available real world datasets close to the ones used in EmbedJoin paper [6]. Table 4.1 summarizes the key characteristics of each dataset. For DNA storage dataset, we use other two datasets. Details about the latter will be provided in Section 4.6.
UNIREF. A dataset of UniRef90 protein sequence data from UniProt project[2]. Each sequence is an array of amino acids in upper case. We kept 400,000 protein sequences of lengths greater than 200.

[1]It is publicly available after free subscription

[2]Available at http://www.uniprot.org

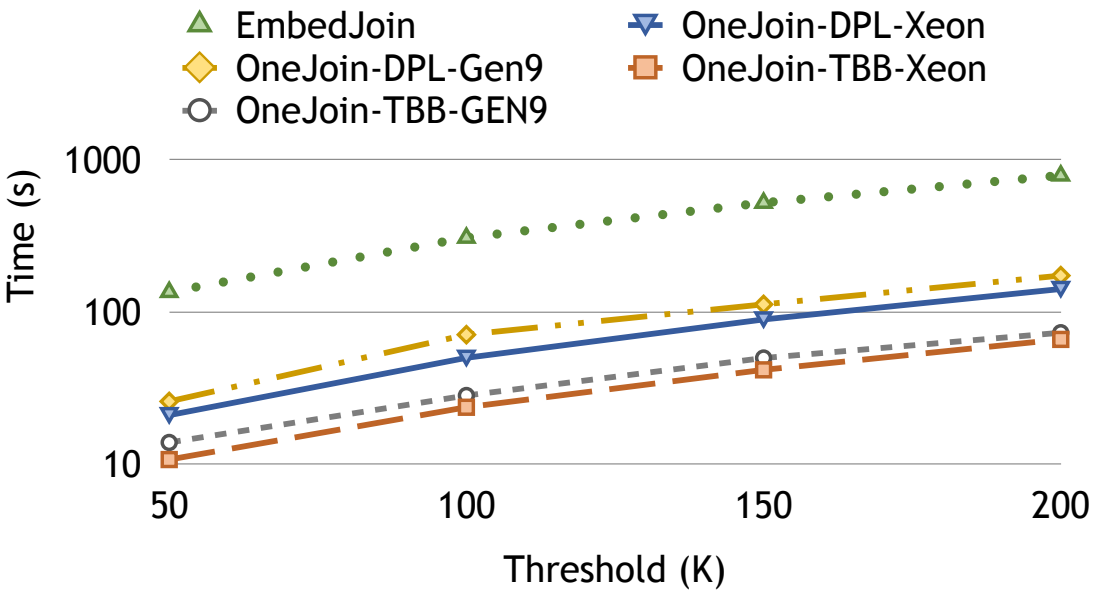


Figure 4.1: Execution time of EmbedJoin and OneJoin under GEN-470KS dataset at various edit distance thresholds.

TREC. A dataset of references from Medline database consisting of titles and abstracts from 270 medical journals[3].
Genomics. Dataset based on Chromosome 20 of 50 individuals obtained from the personal genomes project[4]. It is made by long DNA sequences partitioned in 5000 character long strings. The long DNA sequences are partitioned into shorter sub-strings of length 5,000. We use two variants of this dataset, one with 470k strings (GEN-470KS) and a smaller subset with 20k strings (GEN-20KS).

4.2 Portable Parallelism

In this section we shows the value added by DPC++ as tool to develop software portable across multiple platforms. We start the discussion with a comparison between EmbedJoin - used as reference - and OneJoin. The latter was executed on

[3]Available at http://trec.nist.gov/data/t9_filtering.html
[4]Available at http://personalgenomes.org/

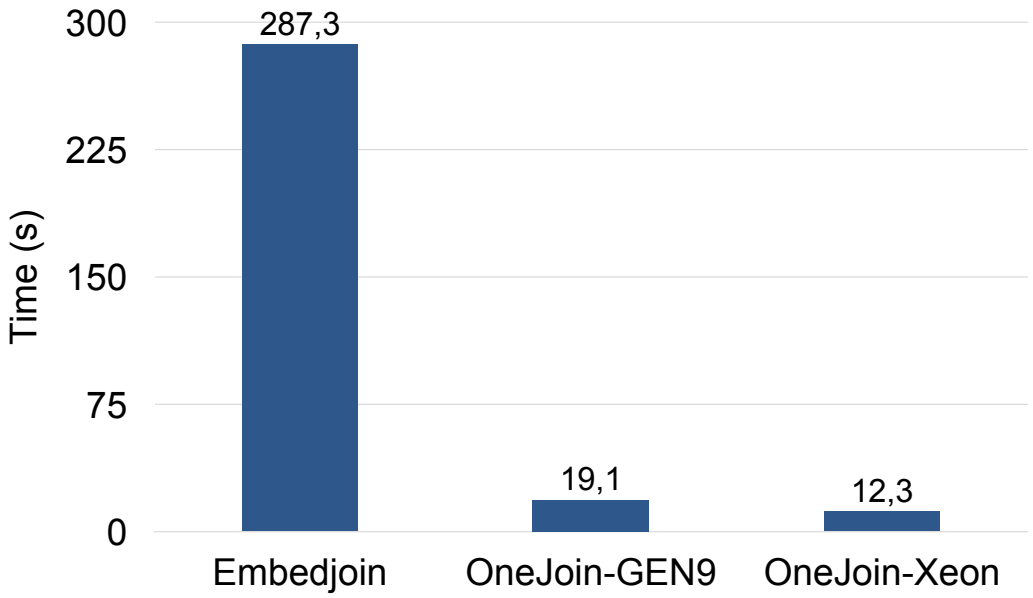


Figure 4.2: Embedding Phase

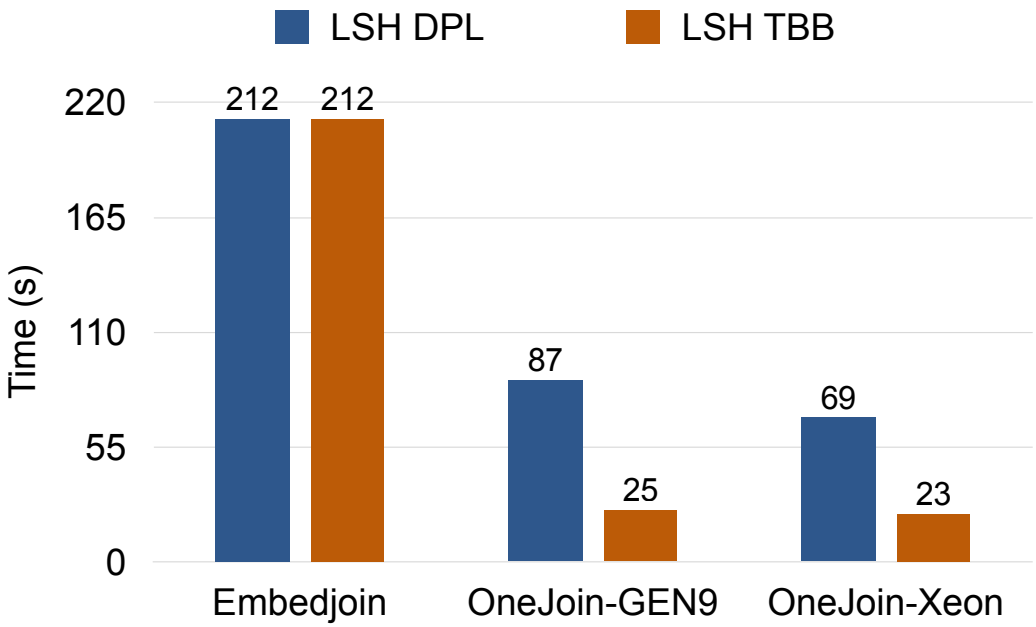


Figure 4.3: LSH Phase

multicore CPU and on integrated Intel Gen9 GPU. The experiment evaluate the algorithms under the `Genomic` dataset (Gen-470KS), our largest dataset. We use

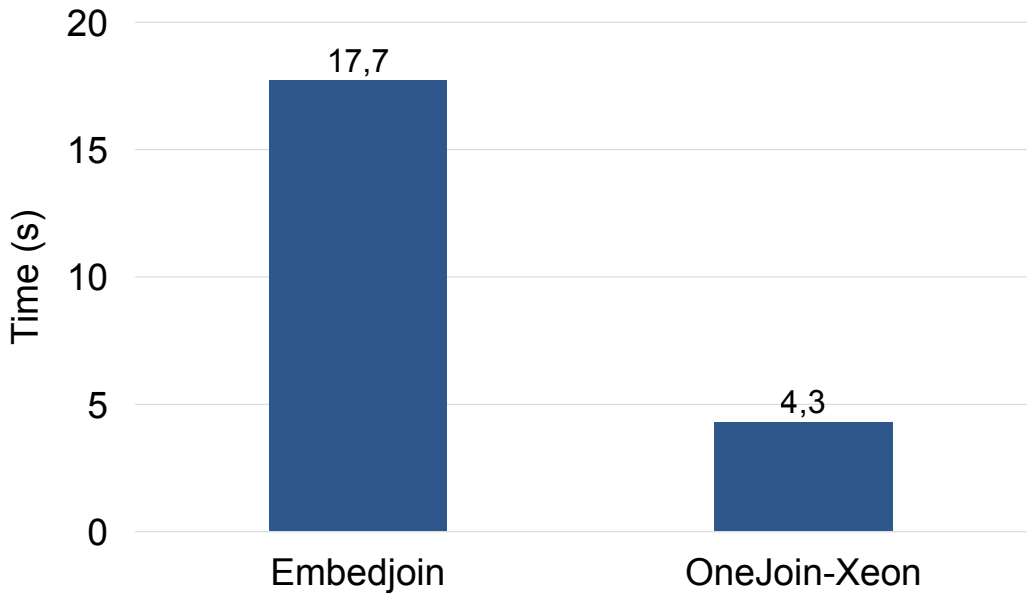


Figure 4.4: Verification Phase

only EmbedJoin as reference, since the other join algorithm failed to execute on our hardware given the dimension of the dataset or took more than 6 hours.

Recall that OneJoin is made by three data-parallel kernel (embedding, bucketization and candidate generation) and several library function call for operation such as sorting, filtering and duplicate elimination. As we mentioned in Section 3, we uses the functions provided by the oneAPI Data Parallel Library (DPL). Since the first version of the code evaluated builds on DPL for such operations, we refer to the first two OneJoin variants evaluated as OneJoin-DPL-Xeon and OneJoin-DPL-GEN9. Notice that we verified the results of our algorithm with the output produced by EmbedJoin for different parameters setting - including the the one used in the benchmark - and the perfect match in all cases guarantees the correctness of our algorithm.

Figure 4.1 shows the total execution time for the algorithm. We can notice that both versions of OneJoin provide a reduction in the total execution time compared to EmbedJoin. In particular OneJoin-DPL-Xeon proves to be $5.5 - 6.5\times$ faster than the sequential algorithm, while OneJoin-DPL-GEN9 provide a $4.5 - 5.5\times$ speedup. Notice that the difference between the two OneJoin variants is only a

different device selector object, that can be chosen at run time. This highlights the advantage of using DPC++ and OneAPI to achieve a portable scalability across processors types.

To understand better the improvement provide by OneJoin, we broke the total time showed in Figure 4.1 for K fixed to 150 in the different contributions of each stage. The execution time per stage is presented in Figures 4.2, 4.3, 4.4. The experiment results show that the main improvement is provided by the embedding stage. Notice that since the verification stage is performed only on CPU only in the GPU case, it is not reported any GPU bar for verification stage. Besides, the improvement at this stage is linear with the number of cores. An important observation concerns the LSH stage, that as showed it does not seem to scale as well as the embedding phase. The explanation is the following. The LSH is made by two data-parallel kernels, that are extremely efficient and that concur to only the 10% of the total LSH stage. The remaining part is lost in data-parallel `operation`, such as sorting or filtering, making the library calls dominating for the 90% of this phase in both CPU and GPU case. The advantage of DPL is that it is can execute basic algorithms (the same provided by the std C++ library) on a device just providing an execution policy coming from the device itself. Thus, changing this execution policy, change the device on which the operation is performed - CPU or GPU in our case. However benchmarking we notice that performances are way worse than the same function implementation provided by another Intel library: Thread Building Blocks (oneTBB). It is a library made exclusively for multi-threading on CPU, and since it is elder than DPL, it showed be much more efficient. Thus, we developed a new version of the OneJoin in which we replaced the DPL library with oneTBB, despite we loose in this way the possibility of executing these common functions on GPU as well. In other words, the new version presents three kernels executing on CPU or GPU and all other operation on CPU multicore through oneTBB functions. We end this section by showing in Figure 4.3, the improvement led by oneTBB to the LSH stage. OneJoin-TBB-Xeon provides a 9× speedup, and OneJoin-TBB-GEN9 provides a 8× speedup over EmbedJoin for the LSH stage. This is translated in an improvement of total time as showed by Figure 4.1. Indeed, 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 the results, all following experiments will consider OneJoin-TBB only.

4.3 Cross-architecture Fork

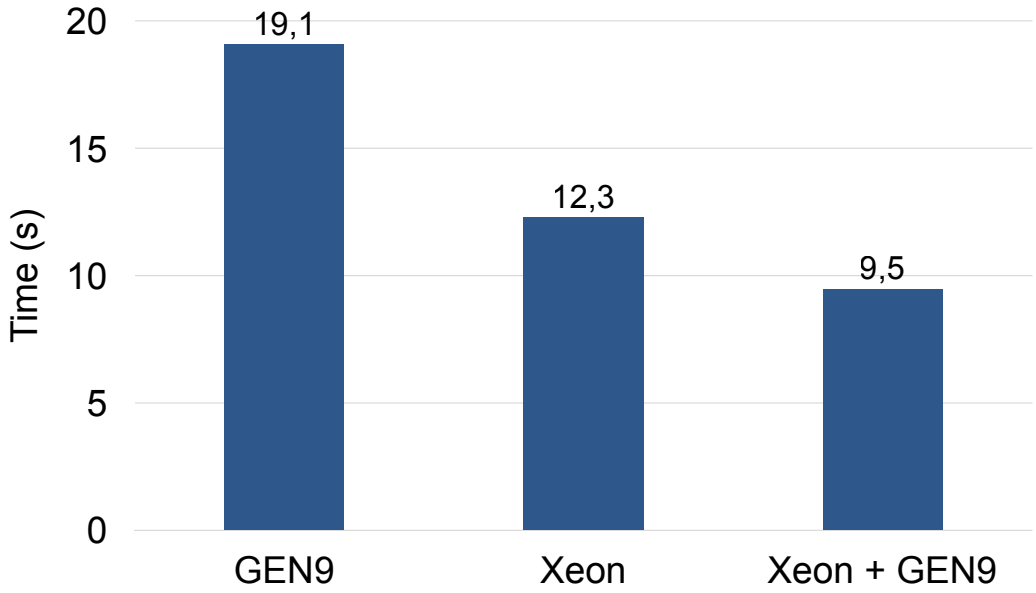


Figure 4.5: Embedding execution time of OneJoin with and without cross-architecture fork

The objective of this section is to show the capability of OneJoin to exploit multiple processor units `simultaneously` by running kernels on multiple devices, i.e. the cross-architecture fork described in Section 3.1. We report the results for embedding kernel only under the Gen-470ks dataset, for a value of edit distance threshold set at 150. The reason is that embedding kernel is one taking more time, and the improvement is clear. Figure 4.5 show the execution time of embedding kernel for three OneJoin configuration: OneJoin-Xeon, OneJoin-GEN9 and OneJoin-Xeon+GEN9. The first two are the CPU-only and iGPU-only configurations as shown in Figure 4.1, while the OneJoin-Xeon+GEN9 is the configuration taking advantage of both devices, with allocation of workload for CPU and iGPU computed on-the-fly. As we see from the bar chart, the cross architecture fork increase

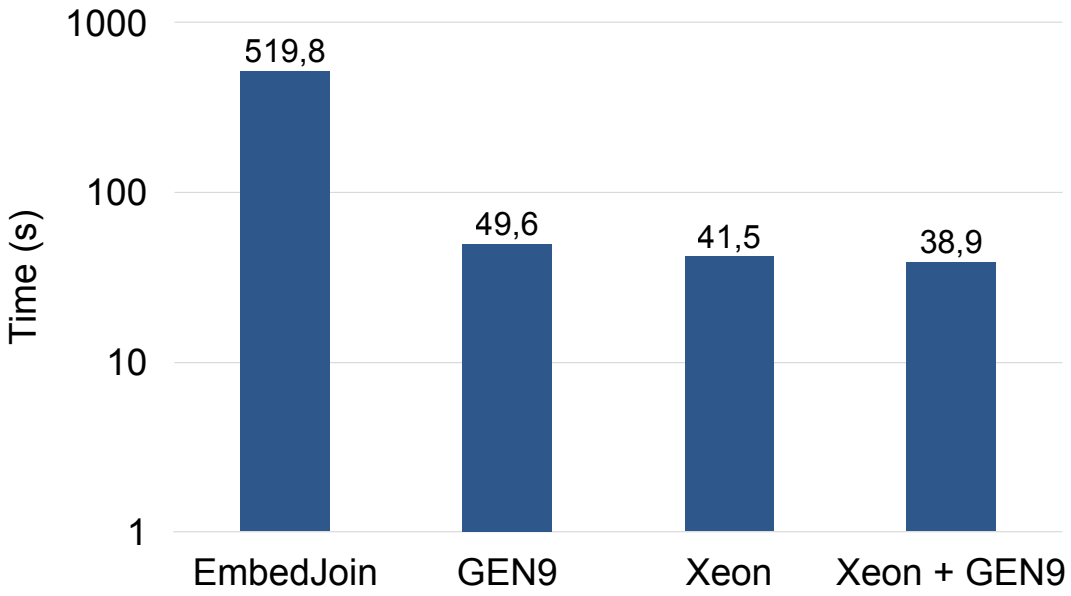


Figure 4.6: Total execution time of OneJoin vs EmbedJoin with and without cross-architecture fork

the speedup of the kernel, as it is 1.3×/2× faster than its CPU-only/GPU-only counterparts.

Moving on Figure 4.6 we can see how this improvement affect the total running time. It is evident that the simultaneously execution on the two devices further widens the gap, leading OneJoin-Xeon+GEN9 to be 13.4× faster than EmbedJoin compared with 10.5× speedup achieved by OneJoin-GEN9, and 12.5× speedup achieved by OneJoin-Xeon. Despite the improvement, comparing Figure 4.5 and Figure 4.6, we can observe that the speed-up in embedding is not completely achieved in the total time of the algorithm as well. The reason, is that the cross architecture fork-join optimize the three kernel only in the fork phase, while at join stage all data parallel operation are computed on multicore CPU only by using oneTBB, included the verification step. So the major presence of CPU in the algorithm mitigates the impact over the total execution time. A second reason is that the Intel Xeon CPU is much faster than the iGPU, thus during the splitting of work, more data are assigned to the CPU to be sure that all devices are busy at same time. This lead the running time to be more close to the GPU time. Although the slight impact on the total running time, this experiment shows the easy with

which DPC++ allows the execution on different processor types.

4.4 Cross-platform parallelism: OneJoin on Discrete GPU

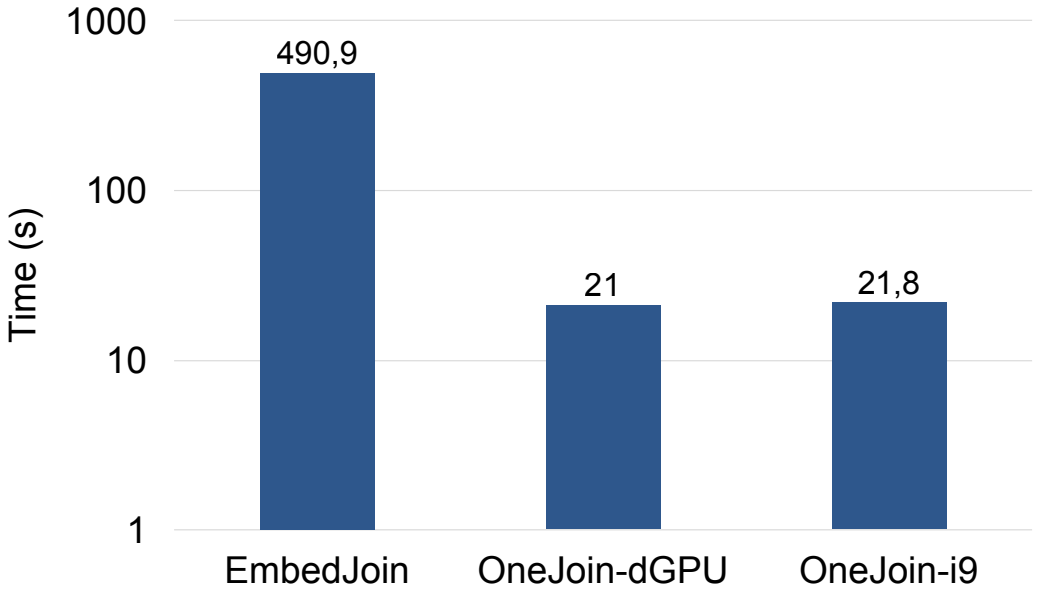
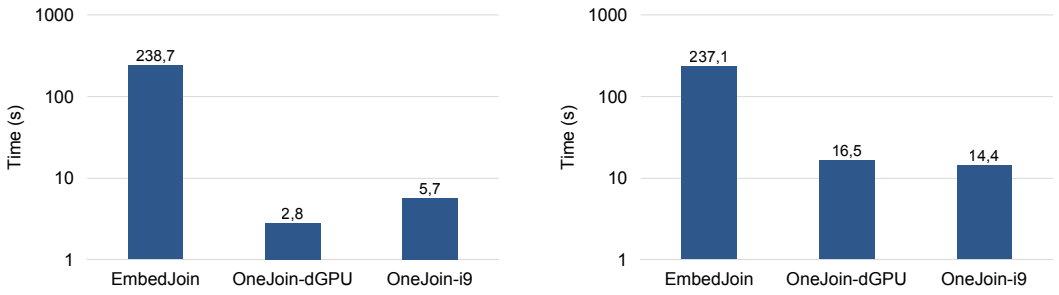


Figure 4.7: Total execution time of OneJoin with discrete GPUs



(a) Embedding execution time

(b) LSH execution time

Figure 4.8: Execution time breakdown of OneJoin with discrete GPUs

So far, we showed the capability of OneJoin (and DPC++) to run the same code

on different devices or to exploit multiple devices simultaneously (cross architecture parallelism). However in all experiments, both CPU and iGPU belonged to Intel family. In this section we evaluate OneJoin on GPU of a different vendor: NVIDIA dGPU. All results in this section come from our local server, equipped with a 12-core CPU and a PCIe-attached, NVIDIA dGPU.

NVIDIA GPU can be programmed by using CUDA. However, thanks to CodePlay's SYCL-for-CUDA extension[5], we can compile a DPC++ program to run on NVIDIA hardware. Notice that this requires no change in term of code, but only a recompilation with a modified Clang++-LLVM compilation infrastructure that supports a CUDA backend. However, the extension for cuda devices is still in a beginning state. For this reason as on date we are not able to exploit the cross-architecture fork in this specific case. Indeed we can chose one backend at runtime when we lunch the code, and if we specify a backed for Intel CPU device NVIDIA devices are not found, and vice versa. So we limit the experiment to 12-core CPU and NVIDIA dGPU separately. This however is sufficient to show the advantages of using DPC++ to run code over hardware of different vendors. We are confident that this limitation will be soon solved, allowing multiple backends.

Figure 4.7 compares performances of OneJoin on 12-core CPU and dGPU under the Gen-470KS dataset. While the three kernels are executed on CPU or dGPU, in both the configuration the join stages (of the fork-join model) uses the CPU-based Intel oneTBB. From Figure 4.7 we can see that OneJoin-dGPU provides a 21× speedup over EmbedJoin. We can also notice that the speedup is comparable with improvement obtained by OneJoin-i9 executed on 12-cores CPU. Instead, comparing dGPU performances with iGPU from the previous results, we can notice that the dGPU is much faster despite the overhead due to the PCI data transfer.

Going further in details, we reports the breakdown across embedding and LSH stages in Figure 4.8. We do not shows the time for verification, since it executed always on CPU, and the improvement is in both cases 12×, as expected having 12-core. Concerning the embedding stage, we can see that using a discrete GPU we obtain a 94× speedup compared to EmbedJoin, while a 45× improvement with the 12-core CPU. Also in this case the impact is minimal on LSH stage that

[5]https://github.com/codeplaysoftware/sycl-for-cuda

shows being only 11-12× faster due to the call to the CPU-based oneTBB library. Again, we highlight that no change in code was necessary to make the algorithm runnable with a different backend, proving the benefit of DPC++ in providing a cross-platform portability.

4.5 Comparison with State-of-the-art Joins

So far, we demonstrated the cross-architecture and cross-platform portability of DPC++. But how does OneJoin performs w.r.t. the other state-of-the-art join algorithm?

In this section we shows macrobenchmark comparing OneJoin with its sequential counterpart - EmbedJoin - and the other join algorithm mentioned in Section 4.1. OneJoin is executed on our local server, and the configuration used is the one exploiting the NVIDIA dGPU, being our best results. We tested the algorithm under three dataset for various edit distance threshold: Trec, Uniref and Gen-20ks described in Section 4.1. Observing Figure 4.9, we can make two considerations. There is a significant difference in performance between the exact algorithm for edit similarity join and algorithm based on the approximation explained in the previous chapters - i.e. EmbedJoin and OneJoin. Both the latter showed to outperform other algorithm; this is especially true for high value of edit distance threshold. This show the effectiveness of low-distortion embedding and LSH to reduce the number of exact edit distance computation, that is the bottleneck of the other algorithm. Second observation is that also in this case OneJoin proves to be up to 3.5× faster than EmbedJoin under TREC, 5.3× under UNIREF, and a 4.3× speedup under Gen20KS. This speedup are lower than the one reported for Gen-470ks dataset, because although this dataset push to the limit the other exact algorithm, the computation is not enough intensive for OneJoin. In other words, the parallel parts is extremely short due to the efficient parallelization of kernels on NVIDIA dGPU, and the majority of the time is spent in initialization and I/O.

4.6 OneJoin for DNA Data Storage

In the last evaluation section, we demonstrate the OneJoin usefulness in the DNA read clustering task. We use two datasets for the experiment. The first one simulated and one obtained from real DNA sequencing. The simulated dataset allows to compute the accuracy of our algorithm in recovering the original reads. It is obtained by loading 1MB into a postgreSQL database. The database has been archived and encoded into 505.783 sequences by means of a tool developed in `pg_oligo_archive` [13]. Each read is 209 nucleotides long. Then using a short-read simulator[6], we emulated sequencer to generate a five million reads from these original oligos containing substitution, insertion, and deletion errors. Finally, we used Accel-Align [14], an accurate, scalable short-read aligner to associate each read to the original oligos. The set of oligos covered by the reads represents for us the ground truth. Notice that in a real-world scenario we cannot use a sequence aligner to recover the original oligos, since the original oligos are still not available at this phase of the recovering process but they are the final objective of the process. We use the ground truth to compare the accuracy of OneJoin-based read consensus solution with Starcode software. Starcode represent the state-of-the-art algorithm for clustering based on edit distance similarity.

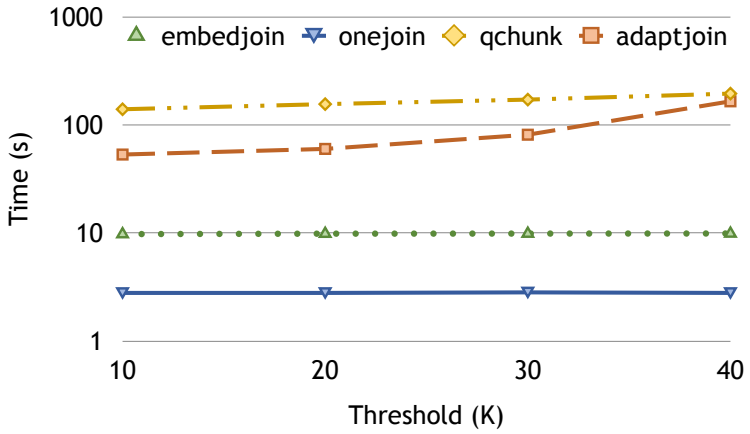
	Execution time	Accuracy (%)
OneJoin	4min 48s	98.3
Starcode	141min 41s	97.9

Table 4.2: Performance and accuracy of OneJoin-based read consensus and starcode.

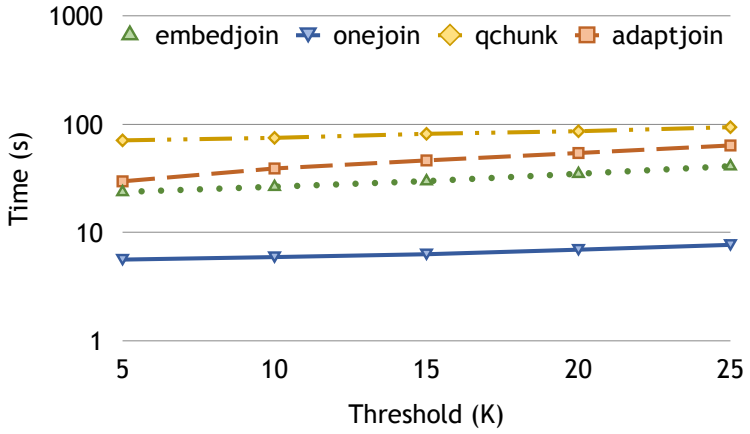
Experiment results are showed in Table 4.2. We compared the execution time of both algorithm and accuracy, computed as $\frac{number-of-oligos-detected-via-consensus}{number-of-oligos-via-alignment}$. Both the program ran on our local server, with Starcode running on 12-core CPU and OneJoin on NVIDIA dGPU. Results indicate that OneJoin outperforms Starcode in terms of running time providing a 29.5× speedup, while both have a comparable accuracy. This proves that OneJoin can compete with state-of-the-art.

[6]https://sourceforge.net/projects/bbmap/

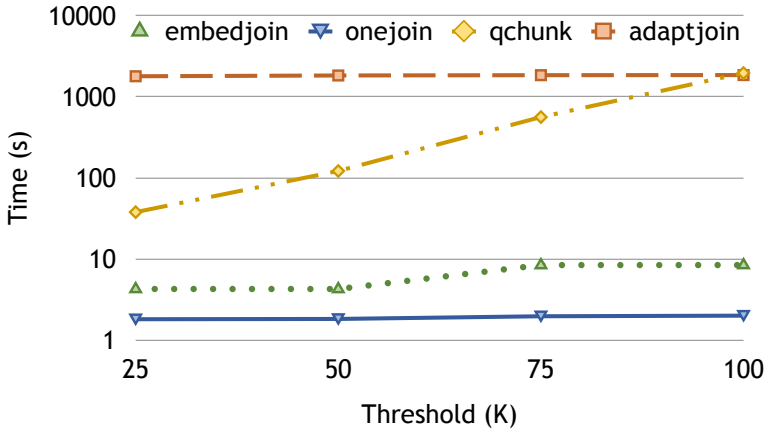
The difference in performances comes from the fact that Starcode, although it uses all the multicores CPU, it performes an exahustive edit distance search and it is not able to exploit the cross-architecture parallelism - running for example on GPU.
Having demonstrate that our algorithm performs well in terms of accuracy, we now shows how it performs in a real-world scenario. We test OneJoin-based clustering solution with Starcode under a dataset obtained from a published experimental study [13]. The dataset consists of 12KB postgreSQL database encoded in 404 oligos. The oligos have been synthesized in a real synthetic DNA and sequenced it back by means of Illumina Novoseq 500 - a short-read, next-generation sequencer. From the sequencing phase we obtained 19 million noise reads, each made by 91 nucleotides. We used the OneJoin-based read consensus solution to recover 403 oligos (out of the original 404). On further inspection we noticed that the missing oligos was covered by only one read, thus it has been classified as noise point and dropped. Despite this, the decoder has been able to recover back the digital data thanks to the use of repetition code inserted during decoding. Concerning the execution time, OneJoin-based clustering solution took only 9 minutes to compute all similar pairs, while the consensus stage and the decoding stage complete in a few seconds. We don't have results for Starcode for this last experiment, since it was not able to finish.
To sum up, we proved with these results that our clustering solution built on OneJoin algorithm - exploiting the portable, cross-architecture parallelism provided by DPC++ - is able to complete the end-to-end data decoding for DNA storage in minutes on a single server compared to the hours taken by the state-of-the-art algorithm.



(a) TREC



(b) UNIREF



(c) GENOMICS 20KS

Figure 4.9: Execution time of join algorithms at various distance thresholds (K)

Chapter 5

Conclusion

In this thesis, we addressed the problem of the edit similarity join and its limitation in terms of scalability when applied to large dataset, as is the DNA data storage case. We developed OneJoin, a data-parallel join algorithm based on DPC++ and OneAPI. We showed how OneJoin can exploit the multiple processor types available on modern heterogeneous computing systems. We proved that our algorithm is able to provide up to 21× speedup compared to EmbedJoin, and up to 29.5× reduction in time in the context of DNA `read consensus`, enable the end-to-end data decoding in minutes.

To our knowledge, OneJoin [1] represents also the first database operator based on DPC++. For this reason we make the code publicly available, in order to encourage further researches on this topic and to extend the DPC++ based implementation to other operators for data analytic engines.

As future work, we are going to address three different points. First, we will investigate the advanced features of DPC++ and oneAPI for kernels optimization, such as the use of ND-Range to express the parallelism and Unified Shared Memory (USM) for the memory management. Secondly, we will extend the portability towards CUDA. The aim is to replace the data-parallel operations such as sorting and reduction with the CUDA library calls, in order to enable the execution on NVIDIA dGPU in place of CPU (oneTBB). Lastly, we will extend the evaluation

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

with further experiments. Specifically, we will evaluate the algorithm on the recently announced Intel DG1 - a new Intel dGPU - and analyze the portability on spatial architectures (FPGA); we will evaluate the capability on OneJoin in performing read-consensus on long-read sequencer, such as Oxford Nanopore.

Bibliography

[1] David Reinsel, John Gantz, and John Rydning. «Data Age 2025: The Evolution of Data to Life-Critical». In: (2017) (cit. on p. 1).

[2] Intel. *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` (cit. on p. 1).

[3] IDC. *Technology Assessment: Cold Storage Is Hot Again — Finding the Frost Point*. `http://www.idc.com/getdoc.jsp?containerId=246732`. 2013 (cit. on p. 1).

[4] SNIA. *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`. 2017 (cit. on p. 1).

[5] V.V. Zhirnov and Daniel Rasic. *2018 Semiconductor Synthetic Biology Roadmap*. Oct. 2018. DOI: `10.13140/RG.2.2.34352.40960` (cit. on p. 2).

[6] Haoyu Zhang and Qin Zhang. «Embedjoin: Efficient edit similarity joins via embeddings». In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2017, pp. 585–594 (cit. on pp. 4, 7, 21, 24, 42).

[7] Diptarka Chakraborty, Elazar Goldenberg, and Michal Kouck. «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*. 2016, pp. 712–725 (cit. on p. 7).

[8] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. «Similarity Search in High Dimensions via Hashing». In: *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB '99. 1999, pp. 518–529 (cit. on p. 8).

[9] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. *Data Parallel C++*. 1st. Apress Open, 2020. ISBN: 9781484255742 (cit. on pp. 13, 17).

[10] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. «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, 1996, pp. 226–231 (cit. on p. 35).

[11] Jiannan Wang, Guoliang Li, and Jianhua Feng. «Can we beat the prefix filtering?: an adaptive framework for similarity join and search». In: *Proceedings of International Conference on Management of Data (SIGMOD)*. 2012, pp. 85–96. URL: http://doi.acm.org/10.1145/2213836.2213847 (cit. on p. 42).

[12] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. «Efficient Exact Edit Similarity Query Processing with the Asymmetric Signature Scheme». In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. Athens, Greece: Association for Computing Machinery, 2011, pp. 1033–1044. ISBN: 9781450306614. DOI: 10.1145/1989323.1989431. URL: https://doi.org/10.1145/1989323.1989431 (cit. on p. 42).

[13] Raja Appuswamy, Kevin Lebrigand, Pascal Barbry, Marc Antonini, Oliver Madderson, Paul Freemont, James MacDonald, and Thomas Heinis. «OligoArchive: Using DNA in the DBMS storage hierarchy». In: *CIDR*. 2019 (cit. on pp. 52, 53).

[14] Yiqing Yan, Nimisha Chaturvedi, and Raja Appuswamy. «Accel-Align: A Fast Sequence Mapper and Aligner based on the Seed–Embed–Extend Method». In: *bioRxiv* (2020). DOI: 10.1101/2020.07.20.211888. URL: https://www.

biorxiv.org/content/early/2020/07/21/2020.07.20.211888 (cit. on p. 52).