1 Background

1.1 DNA, Chromosomes and Genomes

DNA, or *deoxyribonucleic acid*, is a type of molecule that contains all the genetic material found in the cells of all known living organisms [1]. The molecule is composed of two complementary strands of *nucleotides* that are twisted together to form a double helix structure, connected by bonds formed between complementary nucleotides. The two strands are in turn composed of the four nucleotide bases: adenine (A), guanine (G), cytosine (C) and thymine (T), where A and T, and C and G are complementary bases [2, p.15]. Furthermore, the two complementary strands of nucleotide bases actually encode the precise same information. This is because with knowledge of just one of the strands' nucleotide sequence, say $strand_1$, we can determine the sequence of the other strand, $strand_2$, by exchanging each nucleotide in $strand_1$ with their complements and finally reversing the strand to determine what $strand_2$'s sequence is.

Figure 1: A conceptual representation of a DNA molecule made up of two strands. The strands are composed of nucleotides forming base pairs where A (adenine) and T (thymine), and C (cytosine) and G (guanine) are complements of each other.

Relatively small differences in these DNA sequences are what differentiates individuals within the same species from one other. It is therefore interesting to study these sequences of nucleotides encoding organisms' genetic information, as the encoded information can reveal details about both associated physical traits and diseases. In human cells, these DNA strands are estimated to be roughly $3 * 10^9$ bases long [2, p.13].

DNA is organized into structures called *chromosomes*. Humans have 23 chromosome pairs, making up a total of 46 chromosomes. Each of the pairs include one version of the chromosome inherited from the male parent, and one version inherited from the female parent [3].

The term *genome* can be used to refer to the complete genetic material of an organism. In practice, however, the genome of an organism often simply refers to the complete DNA nucleotide

sequence of one set of chromosomes for that organism [2, p.13]. Commonly in bioinformatics, one can also encounter the term *reference genome*, referring to a theoretical reconstruction of an organism's genome created by scientists. Such genome reconstructions are commonly used when examining new DNA sequences, often by aligning new DNA sequences to the reference in order to see at which positions their nucleotides differ and what differences are present at those positions [4].

1.2 Variants and Variant Calling

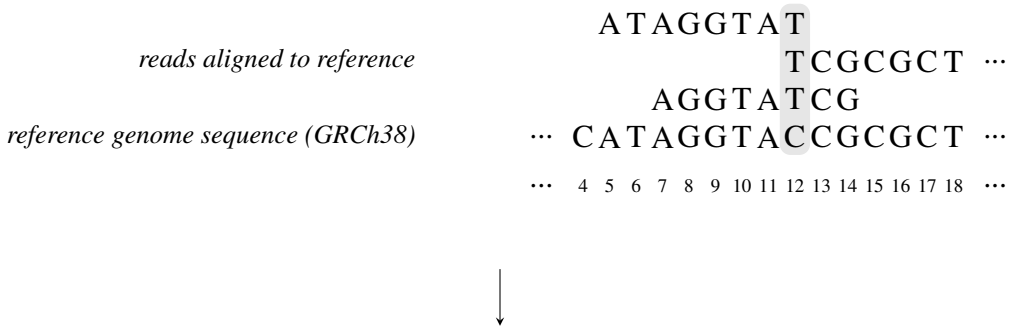
When examining the genome of several individuals within the same species, one will find locations along the genome where the nucleotides differ for the different individuals. These distinct nucleotide manifestations are commonly referred to as *variants*. The term *variant calling* is used to refer to the process of determining which variants an individual has. In other words, given a reference genome sequence, where and how does the genome sequence of the individual of interest differ from the reference sequence. This process can abstractly be described in three steps: 1) sequence the genome of interest to get DNA reads (described in section 1.4), 2) align the reads to the reference genome by finding where along the reference genome sequence each read fits best, usually using a heuristic determining which location the read originates from, and 3) examine the alignments and note where and how the reference and the individual's sequences differ to determine the variants present in the individual's genome [5].

A common way to represent genome sequence variations is to encode them according to the *Variant Call Format* (VCF) file format. The VCF file format encodes a single variant per line, and each line contains a number of columns where each column encodes a particular piece of information about the associated variant, such as [6]:

1. CHROM: an identifier for the reference sequence used, *i.e.* the sequence against which the sequenced reads varies.
2. POS: the position along the reference sequence where it varies against the sequenced reads.
3. ID: an identifier for the variantion.
4. REF: the reference base (or bases) found at the POS position in the reference sequence.
5. ALT: a list of the alternative base (or bases) found at this POS position.

While more columns are usually present, this encapsulated the necessary knowledge about

variants and their representation needed for this thesis.



example.vcf

```
##fileformat=VCFv4.2
##reference=GRCh38
#CHROM POS     ID  REF  ALT  QUAL FILTER INFO
1      12      .   C    T    30   PASS   NS=3;DP=3
...
```

Figure 2: An illustration of how sequenced reads can be aligned against a reference genome sequence in order to call variants present in the genome of the sequenced individual. The called variant in the illustration is then stored in a variant call format (VCF) file where the chromosome identifier, 1-indexed position along the chromosome reference sequence, an identifier for the variant, the allele or alleles present in the reference sequence, the alternative variant allele or alleles, along with a number of other parameters are used to represent each variant.

1.3 Genotype and Genotyping

The term *genotype* refers to the set of variants an individual carries at a particular location along the genome sequence of all of its chromosomes [7]. For humans, who have two of each chromosome, a genotype would refer to two variants, one in each of the two chromosomes. *Genotyping* an individual refers to the process of determining which genotypes an individual carries. In most genotyping software tools today, genotypes are given in a format that specifies whether a particular variant is present in none, one or both of a human's chromosomes. For instance, given a reference genome sequence where a variant site is known to could manifest an A at a particular allele where the reference sequence contains a C, a human individual's genotype for this variant could either be referred to as $0/0$, meaning that the variant is present in neither of the chromosomes, $0/1$, meaning that the variant is present in one of the chromosomes, or $1/1$, meaning that the variant is present in both chromosomes.

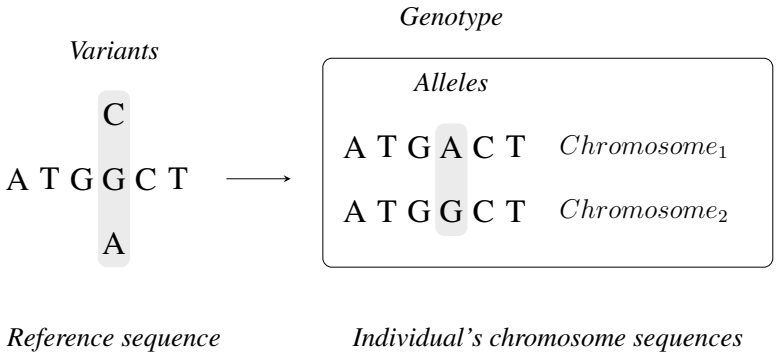


Figure 3: In humans, where there are two chromosomes, a genotype constitutes as a set of two alleles, one in each chromosome at the variant location. Along the reference sequence on the left, several possible variants may be known to occur at a specific location. After examining the sequence of an individual, we try to determine the individual's genotype by scoring which variants are present in each chromosome at the location of interest.

The most established way to genotype an individual today is to align DNA reads to a reference genome sequence and then examine how the reads differ from the reference sequence to determine which variants are present, and which genotypes are most probable at the different locations [4]. However, given how many reads one have come to expect from high-throughput sequencing today [1.4] and how time consuming it is to align reads to a $3 * 10^9$ long reference sequence, although accurate, this strategy is very compute- and time consuming. A new prominent strategy has emerged in recent years that helps to alleviate the compute- and time consumption aspect of genotyping. Statistics based methods, usually referred to *alignment-free* genotyping methods, where the variant calling step where reads are aligned to the reference genome is skipped altogether. In such methods, small parts of the sequenced reads called *k*mers are analyzed, and bayesian models are then used to determine which genotypes are most probable given the results from the *k*mer analysis and previous knowledge accumulated over years of research [8, 9, 10]. One such bayesian genotyper, KAGE, have recently showed that it can genotype a human individual more than 10 times faster than any other known genotyper tool, while still providing competitive accuracy scores [8].

1.4 High-Throughput DNA sequencing

High-throughput sequencing (HTS), also known as *next-generation sequencing* (NGS), refers to an assortment of recently developed technologies that parellalize the sequencing of DNA fragments to provide unprecedented amounts of genomic data in short amounts of time. While several such technologies with varying details exist today, they commonly follow a general paradigm of performing a template preparation, clonal amplification where they clone pieces of DNA in order to sequence the clones in parallel, and finally cyclical rounds of massively

parallel sequencing [11]. The resulting DNA sequences produced by HTS technologies are usually referred to simply as (DNA) *reads*. Such reads are commonly stored as plain text in FASTA or FASTQ files, which can later be used for different kinds of analysis such as genotyping. Depending on which HTS technology is used, one can expect read lengths ranging from as low as 150 bases using *Illumina* technologies, referred to as *short reads* [12], all the way up to 15-20 thousand bases using *Pacific Biosciences* (PacBio) technologies, referred to as *long reads* [13]. Three factors are important to consider when determining which HTS technology to use for a given purpose: 1) the read lengths produced, 2) the average probability for each base being erroneous, usually referred to as the error rate, and 3) the cost of sequencing given the technology, which can potentially limit how much data one may be able to produce.

1.5 *k*mers and the *k*mer Counting Problem

...

1.6 Graphical Processing Units

Graphical Processing Units (GPUs) are massively parallel processing units designed for high-throughput parallel computations. This is as opposed to *Central Processing Units* (CPUs), which are designed to quickly perform many serial computations. GPUs were originally developed to accelerate computations performed on images, a highly parallel task where it is commonplace to have millions of relatively small independent computations that must be performed quickly in a single memory buffer. Although GPUs have mainly been used for graphical computations, they have in recent years been adopted in other areas as well with the introduction of the *General Purpose Graphical Processing Unit* (GPGPU) (**GPGPU cite**). The concept of the GPGPU is to use a GPU, which is designed for computer graphics, to perform computations in other domains where CPUs are typically used. Fields such as artificial intelligence (**AI accelerated by GPU cite**) and the broader scientific computing have enjoyed great utility from GPUs, using them to accelerate embarassingly parallel problems, *e.g.*, matrix operations. Despite being similar in power consumption, a GPU can provide much higher instruction throughput and memory bandwidth compared to its CPU competitors. These capability advantages exist in GPUs because they were specifically designed to perform well with regards to these dimensions.

As of 2023, Nvidia control the vast majority of the GPU market share, with only *Advanced Micro Devices* (AMD) and Intel as current serious competitors (**try to find a serious cite**). Fur-

thermore, Nvidia GPUs with their CUDA programming model is considered to be the standard for scientific computing today (**cite**). Although most of the GPUs manufactured by different GPU manufacturing companies are very similar in both architecture and compute models, the term *GPU* will for the remainder of this thesis specifically refer to Nvidia GPUs, as the work presented in this thesis was developed and tested using only Nvidia GPUs.

1.6.1 GPUs in Computers

Two main computer GPU setups are prominent today: *integrated* graphical processing units (iGPUs), and *discrete* graphical processing units (dGPUs). iGPUs are GPUs integrated onto the same die as a computer's CPU, where the two share the same physical *Random Access Memory* (RAM) unit. dGPUs are dedicated GPU devices that are physically distinct from the host computer's CPU and RAM, and have their own physical RAM. dGPUs are significantly more powerful in terms of compute throughput when compared to iGPUs. However, having their own physical RAM introduces an overhead; Memory buffers with input data have to be copied to the dGPU's RAM before processing, and results have to be copied back from the dGPU's RAM to the host RAM.

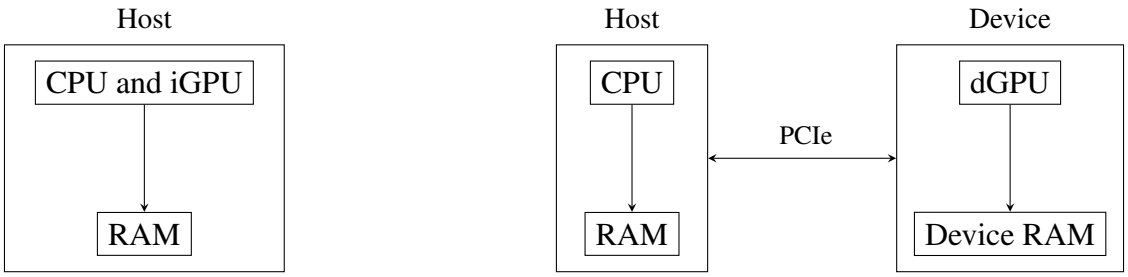


Figure 4: **Left**: A computer setup with a CPU and an iGPU sharing the same die and the same physical RAM. **Right**: A computer setup where a dGPU is connected over PCIe and the dGPU has its own physical RAM, adding the overhead of copying data both to and from the host computer when utilizing the GPU.

For the work presented in this thesis, only dGPUs were utilized. Therefore, the term GPU will from here on out be referring to a dGPU and not an iGPU. This means that all GPU implementations discussed in this thesis will include copying memory back and fourth from the *host* (CPU) RAM and the *device* (GPU) RAM. It is also possible for a single computer to have several connected GPUs, allowing for further parallelization of both memory transfers and compute, however this was not utilized in this thesis' work.

1.6.2 Programming Model and CUDA

Modern GPUs can in effect be considered to be massive *Single Instruction Multiple Data* (SIMD) machines. Strict flow control is therefore important; The same set of instructions should run in the same order for maximum utilization of the GPU's capability.

...

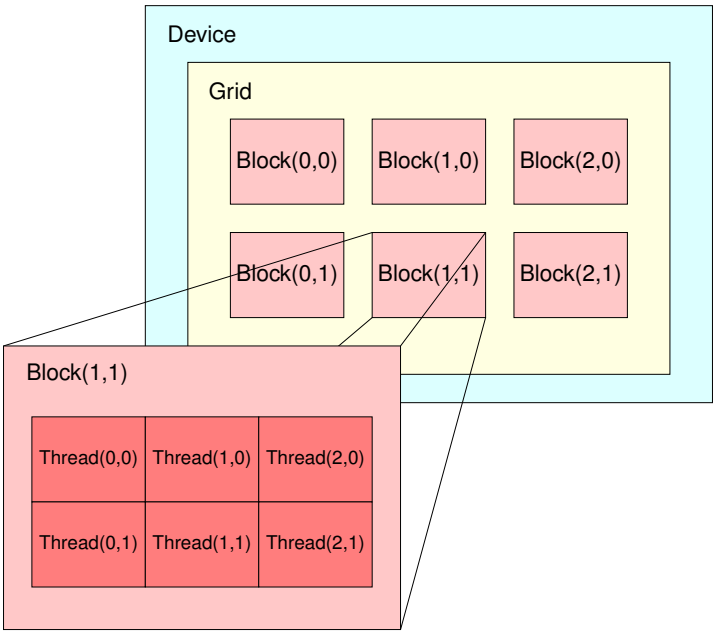


Figure 5: An overview of the CUDA programming model:

1.7 NumPy

NumPy is a scientific computing library for Python that provides support for fast multi-dimensional arrays along with a multitude of mathematical and other types of functions to operate on arrays efficiently [14]. NumPy works as a Python interface to fast C and C++ code that implements the underlying functionalities. This underlying code relies on vectorization and SIMD instructions to perform array operations fast. While NumPy's standard functionality is designed to run efficiently on a single CPU core, multithreading can be utilized to both parellalize on the local data (SIMD) and the total work level (multithreading) at the same time. Its flexible and easy-to-use interface along with its highly performant solutions that supports a wide range of hardware has made it a popular choice for any array-based scientific computing in Python.

1.8 CuPy

CuPy is GPU accelerated NumPy [14] and SciPy [15] compatible array library that, much like NumPy [14], provides a multi-dimensional array object as well as mathematical functions and routines to operate on these arrays. In fact, CuPy's interface is designed to closely follow that of NumPy, meaning that most array-based code written in NumPy can trivially be replaced with CuPy to GPU accelerate the array operations. CuPy, unlike NumPy, will store all array data in GPU memory and all array routines will be performed by the GPU.

1.9 Nucleotide Binary Encoding

DNA nucleotide sequences (described in section 1.1) inside computer software is commonly represented simply by a sequence of the 8 bit characters A, C, T and G (or alternatively the lowercase a, c, t and g). This representation, however, is cumbersome to operate on and requires large amounts of memory to store. To circumvent these issues, a widely adopted technique is to encode the nucleotides into binary form. This leads to much quicker processing of nucleotide sequences and reduces the memory usage needed to store the sequences by 75%. This is achieved by realizing that only 2 bits, giving $2^2 = 4$ possible unique states, is enough to represent all of the four DNA nucleotides A, C, G and T. The binary encoding can be extended further to represent whole nucleotide sequences in binary arrays. For example, an integer array, if interpreted 2 consecutive bits at a time, can represent such a sequence.

Nucleotide to 2 bit encoding lookup table

A	⟷	00
C	⟷	01
G	⟷	10
T	⟷	11

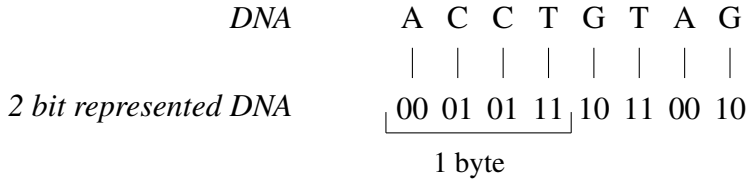


Figure 6: A lookup table showing how nucleotides can be encoded using 2 bits and a DNA nucleotide sequence represented both as plain characters as well as its 2 bit encoded representation. Recall that computers use 8 bits to represent a single nucleotide with a character, whilst the 2 bit encoding only needs 2 bits to represent a nucleotide.

2 Methods

In this section we describe how GPU acceleration support was provided for the KAGE genotyping pipeline, resulting in GKAGE - a version of KAGE where parts of the pipeline is GPU accelerated.

When developing GKAGE, we started with KAGE as a baseline and analysed the pipeline to find the most pronounced bottlenecks in terms of runtime. We then examined whether these bottlenecks could benefit from GPU acceleration by taking into consideration the type of computations that were performed and whether they would be suitable for the GPU architecture as well as how much of the overall runtime the component constituted. We would then intuit whether the component in question would be worthwhile trying to GPU accelerate given the project's time constraints and estimated difficulty.

The baseline KAGE pipeline is split into two distinct processes: 1) counting the *k*mer frequencies observed in the DNA reads of the individual being genotyped, and 2) genotyping the individual using a statistical model based on the observed and expected *k*mer frequencies. Initial benchmarking on a consumer desktop revealed that the *k*mer counting step constituted more than 96% of the total KAGE runtime, with *k*mer counting taking 2080 seconds and the genotyping step taking 70 seconds for a total of 2150 seconds (35 minutes and 50 seconds) to genotype a full human genome. Thus, the *k*mer counting step, having such a clear margin for runtime improvement over the genotyping step, was the initial focus for GPU acceleration for potential runtime speedup.

2.1 GPU Accelerating *k*mer Counting

While different GPU accelerated *k*mer counting solutions such as Gerbil [16] have been developed in previous work, we opted to develop our own *k*mer counting method because the *k*mer counting problem solved in KAGE is slightly different than the typical *k*mer counting problem described in section 1.5. Rather than counting the frequency of every observed *k*mer in the input reads, or even the frequency of every observed *k*mer where the frequency is larger than some threshold, KAGE is only interested in counting the observed frequencies for a predetermined set of *k*mers. This revised problem is easier to solve in practice because the memory constraints are significantly less. In the interest of brevity, we will refer to the typical *k*mer counting problem described section 1.5 as *full kmer counting*, and the simpler problem where we only count the frequencies of a predetermined set of *k*mers as *partial kmer counting* 7.

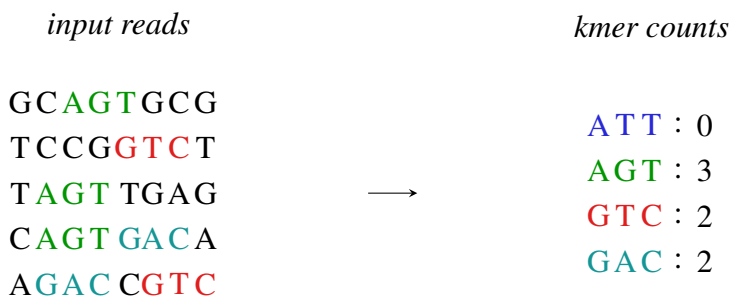


Figure 7: In KAGE, we are only interested in counting the observed frequencies of a predefined set of *k*mers, as opposed to every observed *k*mer in the sequenced reads.

2.1.1 Hash Table Counter based on CuPy

The Python package npstructures [17] contained a counter object designed to count a static set of *k*mer's frequencies. This counter object was written in NumPy and implemented a hash table that effectively utilized NumPy's fast array operations to count *k*mers in chunks. The underlying hash table was based on npstructure's ragged array implementation, which is an effective, NumPy based, two-dimensional array structure where the column lengths can vary. The array column lengths (bucket sizes) were determined during initialization based on how many of the *k*mers from the unique *k*mer set hashed to that particular bucket. Because this counter object was implemented using NumPy's array functionality, it served as a good candidate for GPU acceleration where NumPy's CPU vectorization could potentially be manyfold improved by the massively parallel GPU.

In order to GPU accelerate npstructure's counter object we utilized the fact that CuPy's interface is remarkably similar (by design) to that of NumPy. In fact, CuPy could be described as a GPU accelerated subset of NumPy. This, in addition to the fact that npstructure's counter object was implemented almost entirely using NumPy, allowed for swift GPU acceleration in a way where we hardly had to write any new code and without having to leave Python at all.
./. 2.bin.

(explain how to replace numpy with cupy)

mypackage.my_funcs.py

```
import numpy as np


def some_func_using_numpy():
```

```
  return np.zeros(1000)
```

mypackage.my_classes.py

```
import numpy as np

class SomeClassUsingNumPy:
  def __init__(self):
    self.data = np.zeros(1000)

  def get_data(self):
    return self.data
```

mypackage.__init__.py

```
from my_funcs import some_func_using_numpy
from my_classes import SomeClassUsingNumPy

# Swaps NumPy with lib (presumably CuPy)
def set_backend(lib):
  from . import my_funcs
  my_funcs.np = lib

  from . import my_classes
  my_classes.np = lib
```

program.py

```
import cupy as cp

import mypackage
mypackage.set_backend(cp)

array = mypackage.some_func_using_numpy()
type(array) # cupy.ndarray
```

2.1.2 Parallel GPU Hash Table implemented in native CUDA

...

2.1.3 Parallel GPU Hash Table without C++

...

2.2 GPU Accelerating *k*mer Hashing

References

[1] National Human Genome Research Institute. *Deoxyribonucleic Acid (DNA) Fact Sheet*. https://www.genome.gov/about-genomics/fact-sheets/Deoxyribonucleic-Acid-Fact-Sheet. Accessed: 2023-03-30. 2023.

[2] Gautam .B Singh. *Fundamentals of Bioinformatics and Computational Biology. Methods and Exercises in MATLAB*. Springer, 2015.

[3] National Human Genome Research Institute. *Chromosomes Fact Sheet*. https://www.genome.gov/about-genomics/fact-sheets/Chromosomes-Fact-Sheet. Accessed: 2023-03-31. 2023.

[4] Aaron McKenna et al. "The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data". en. In: *Genome Res* 20.9 (July 2010), pp. 1297–1303.

[5] Stepanka Zverinova and Victor Guryev. "Variant calling: Considerations, practices, and developments". en. In: *Hum Mutat* 43.8 (Dec. 2021), pp. 976–985.

[6] The International Genome Sample Resource. *Encoding Structural Variants in VCF (Variant Call Format) version 4.0*. https://www.internationalgenome.org/wiki/Analysis/Variant%20Call%20Format/VCF%20(Variant%20Call%20Format)%20version%204.0/encoding-structural-variants. Accessed: 2023-04-7. 2023.

[7] National Human Genome Research Institute. *Genotype*. https://www.genome.gov/genetics-glossary/genotype. Accessed: 2023-03-31. 2023.

[8] Ivar Grytten, Knut Dagestad Rand, and Geir Kjetil Sandve. "KAGE: Fast alignment-free graph-based genotyping of SNPs and short indels". In: *bioRxiv* (2021). DOI: 10.1101/2021.12.03.471074. eprint: https://www.biorxiv.org/content/early/2021/12/20/2021.12.03.471074.full.pdf. Address <https://www.biorxiv.org/content/early/2021/12/20/2021.12.03.471074>.

[9] Luca Denti et al. "MALVA: Genotyping by Mapping-free ALlele Detection of Known VAriants". In: *iScience* 18 (2019). RECOMB-Seq 2019, pp. 20–27. ISSN: 2589-0042. DOI: https://doi.org/10.1016/j.isci.2019.07.011. Address <https://www.sciencedirect.com/science/article/pii/S2589004219302366>.

[10] 1000 Genomes Project Consortium et al. "A global reference for human genetic variation". In: *Nature* 526.7571 (2015), p. 68.

[11] Jason A Reuter, Damek V Spacek, and Michael P Snyder. "High-throughput sequencing technologies". en. In: *Mol Cell* 58.4 (May 2015), pp. 586–597.

[12] Illumina. *Read length recommendations*. https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/read-length.html. Accessed: 2023-04-9. 2023.

[13] M. Mahmoud et al. "Utility of long-read sequencing for All of Us". In: *bioRxiv* (2023). DOI: 10.1101/2023.01.23.525236. eprint: https://www.biorxiv.org/content/early/2023/01/24/2023.01.23.525236.full.pdf. Address <https://www.biorxiv.org/content/early/2023/01/24/2023.01.23.525236>.

[14] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. Address <https://doi.org/10.1038/s41586-020-2649-2>.

[15] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[16] Marius Erbert, Steffen Rechner, and Matthias Müller-Hannemann. "Gerbil: a fast and memory-efficient k-mer counter with GPU-support". In: *Algorithms for Molecular Biology* 12.1 (Mar. 2017), p. 9.

[17] Knut Dagestad Rand and Ivar Grytten. *npstructures*. https://github.com/bionumpy/npstructures. 2022.