UNIVERSITY
OF OSLO

Master's thesis

Speeding up Genotyping through GPU Acceleration

Jørgen Wictor Henriksen

Programming and System Architecture

60 ECTS study points

Department of Bioinformatics

Faculty of Mathematics and Natural Sciences

Spring 2023

Jørgen Wictor Henriksen

Speeding up Genotyping through GPU Acceleration

Supervisors:

Ivar Grytten

Knut Rand

Geir Kjetil Sandve

Abstract

In the last couple of decades, high-throughput sequencing has steadily become more effective and orders of magnitude cheaper. With the potential for millions of genomes being sequenced in the coming years, tools for analysing the large amounts of sequenced data will become increasingly important. Recent work in alignment-free genotyping methods have shown that alignment-free methods where we use statistical methods on analysis of *k*mers from sequenced reads can give competitive accuracies while being significantly faster compared to more established alignment-based methods. A recently published genotyper, KAGE, showed that an alignment-free genotyper implemented in Python could yield competitive accuracies while being more than 10 times faster than any other known method. This thesis explores how parts of KAGE that deals with large matrix- and array-operations can be GPU accelerated, and finally presents GKAGE, a GPU accelerated version of KAGE. GKAGE achieves up to 10 times speed up compared to KAGE and is able to genotype a human individual in only a few minutes on consumer grade hardware.

Contents

1 Introduction

...

2 Background

2.1 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.

$$\begin{array}{lccccccccc}
 & & & & & \textit{DNA} & & & & \\
\textit{strand}_1 & \cdots & \mathrm{T} & \mathrm{A} & \mathrm{C} & \mathrm{C} & \mathrm{G} & \mathrm{A} & \mathrm{C} & \cdots \\
 & & | & | & | & | & | & | & | & \\
\textit{strand}_2 & \cdots & \mathrm{A} & \mathrm{T} & \mathrm{G} & \mathrm{G} & \mathrm{C} & \mathrm{T} & \mathrm{G} & \cdots
\end{array}$$

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].

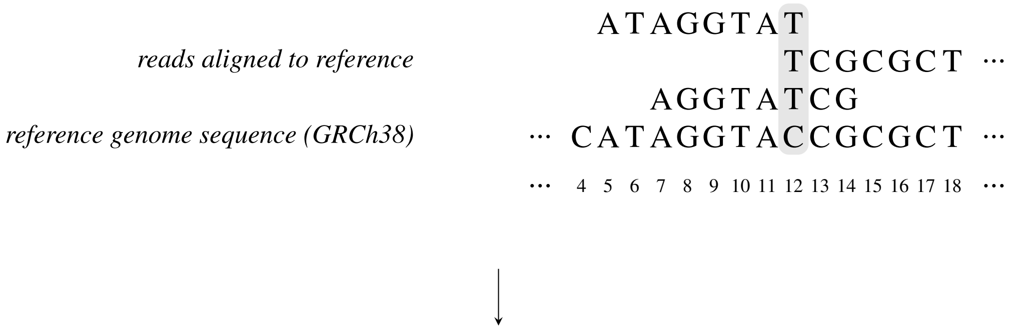
2.2 Variants and Variant Calling

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 2.4), 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.

2.3 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.

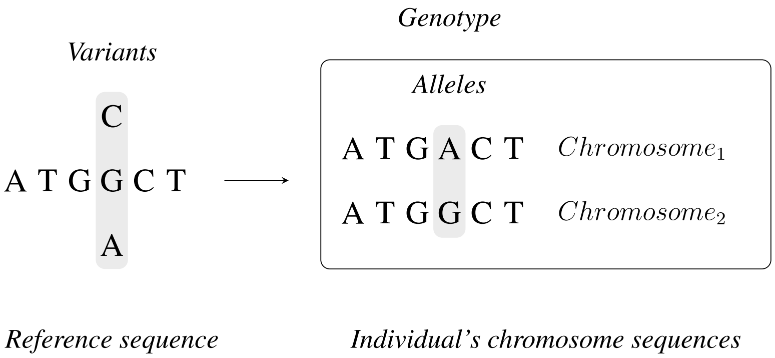


Figure 3: In humans, where there are two chromosomes, a genotype constitutes as a set of two alleles, one in each chromosome at the variant location. Along the reference sequence on the left, several possible variants may be known to occur at a specific location. After examining the sequence of an individual, 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 [2.4] 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].

2.4 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.

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

A *k*mer is a substring of *k* consecutive nucleotides that occur in a DNA (or RNA) sequence. Because of how single nucleotides can be represented in a computer's memory using only 2 bits 2.13, and how when we sequence an individual's genome we can not know which strand our sequenced read comes from, a popular choice of value for *k* is 31 - the default value used in KAGE 2.12. The value 31 is used in KAGE for two reasons: 1) having an odd value for *k* ensures that no *k*mer is equal to its reverse complement, and 2) having *k* equals 31 means we need $31 * 2 = 62$ bits to represent the *k*mer in a computer's memory, thus the *k*mer will fit inside a single 64-bit integer.

A common problem in various bioinformatics applications is to count the number of times each valid *k*mer in a set of nucleotide sequences occur in those sequences. This problem is commonly refered to as *k*mer counting.

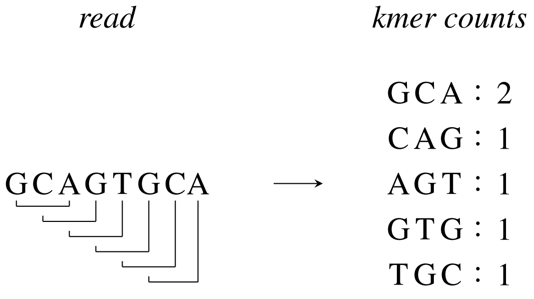


Figure 4: *Full* *k*mer counting where we count the observed frequency of every valid *k*mer in our read set.

The genotyping software tool KAGE, detailed in section 2.12, contains *k*mer counting as one of its core steps in its genotyping pipeline. However, the *k*mer counting process in KAGE is slightly different from the process commonly refered to by the term *k*mer counting. Rather than counting the observed frequency of every valid *k*mer in a set of input reads, KAGE only counts the observed frequencies of every *k*mer in a predefined set. Given how many valid *k*mers one can observe in a set of (hundreds of?) millions of reads (**cite?**), which is typical when sequencing and genotyping a human genome, not needing to store each of these with an associated count value makes this new *k*mer counting variant significantly less memory and time consuming.

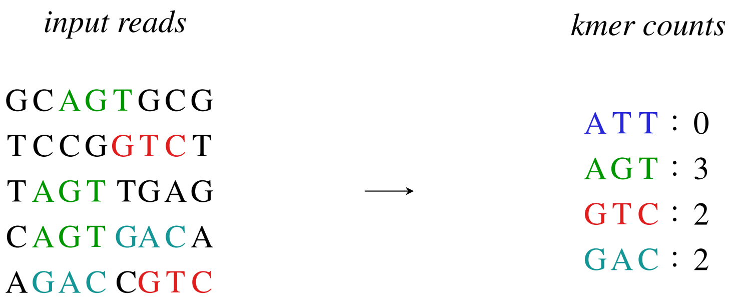


Figure 5: *Partial k*mer counting where we only count the observed frequencies of *k*mers present in a predefined set. In this example, our set of predefined *k*mers is {ATT, AGT, GTC, GAC}. During counting, *k*mers not present in this set are skipped.

Henceforth in this thesis, we will in the favour of brevity refer to the former *k*mer counting process where we count every valid *k*mer's occurence as *full k*mer counting, and to the latter process where we only count the occurences of *k*mers in a predefined set as *partial k*mer counting.

While several *k*mer counting software tools have been developed in previous work, with at least one, Gerbil [14], having support for GPU acceleration [15], these tools are designed to solve the *full k*mer counting problem (**haven't checked this, would probably take a while**).

2.6 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**). Furthermore, 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.

2.6.1 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.

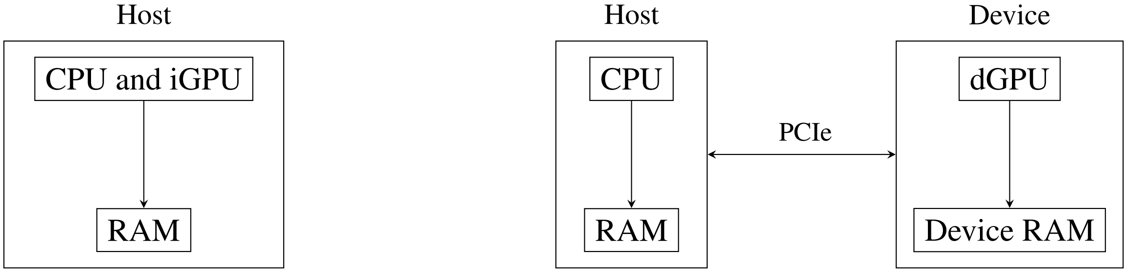


Figure 6: **Left**: A computer setup with a CPU and an iGPU sharing the same die and the same physical RAM. **Right**: A computer setup where a dGPU is connected over PCIe and the dGPU has its own physical RAM, adding the overhead of copying data both to and from the host computer when utilizing the 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.

2.6.2 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.

...

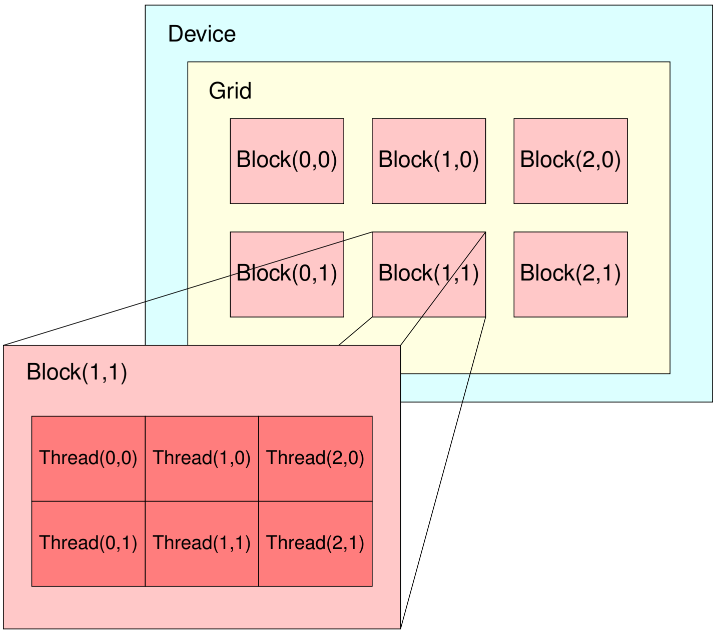


Figure 7: An overview of the CUDA programming model:

2.7 pybind11

pybind11 [16] is a C++ library that provides easy-to-use macros and tools for creating bindings between Python and C++. Its main use case is to create Python bindings for existing C++ code. Using pybind11, this can be achieved seamslessly by using C++ macros to expose C++ functions, classes and their methods to Python. In addition, pybind11 also allows for direct use of certain Python types such as lists, tuples and dicts in C++, supporting both receiving and returning references to such Python objects in C++. NumPy is also supported, making it easy to send NumPy array references to C++ functions where the C++ function is allowed the freedom to both read and write to the array's data. While the work in this thesis evolves aroud GPU accelerating parts of a software tool written in Python, some of the solution implementations presented in this thesis were written partly in C++ for fast performance. pybind11 were in these cases crucial in order to create the bindings from these C++ implementations, making them usable directly in Python.

2.8 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 [17]. 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.

2.9 CuPy

CuPy is GPU accelerated NumPy [17] and SciPy [18] compatible array library that, much like NumPy [17], 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.

2.10 NumPy Structures

NumPy Structures (npstructures) is a Python package built on top of NumPy that provides data structures with NumPy-like features to augment the NumPy library [19]. This is achieved by building these new data structures using NumPy's underlying multi-dimensional array object and fast array routines.

Some of NumPy Structures' data structures have been central in work done in this thesis. Those data structures will therefore be detailed in this section.

2.10.1 Ragged Array

A central feature in NumPy Structures is its ragged array object, a two-dimensional array data structure with differing column lengths that provides NumPy-like behaviour and performance. The ragged array object works as a drop in alternative to NumPy's multi-dimensional array object where one needs an array structure where the column lengths can vary, supporting many of the common NumPy functionalities such as multi-dimensional indexing, slicing, ufuncs and a subset of the function interface.

```
>>> import numpy as np
>>> from npstructures import RaggedArray
>>> data = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> column_lengths = np.array([4, 2, 3])
>>> ra = RaggedArray(data, column_lengths)
>>> ra
ragged_array([0, 1, 2, 3]
[4, 5]
[6, 7, 8])
>>> ra.ravel()
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> type(ra.ravel())
<class ’numpy.ndarray’>
>>> np.sum(ra)
36
```

Figure 8: A simple illustration of how NumPy Structures' data structures can be used directly in Python as drop-in augmentations to the NumPy library.

2.10.2 Hash Table and Counter

NumPy Structures also provides a memory efficient hash table built on top of the ragged array data structure. This hash table is designed to give dictionary-like behaviour for NumPy-arrays, meaning chunks of key-value pairs can be operated on at once using fast NumPy array routines. This hash table object is in turn the base for a counter object that allows for counting of occurences of a predefined set of keys.

The counter (built on top of the hash table) achieves its memory efficiency by implementing a type of bucketed hash table where a ragged array is used to represent the table, and the number of rows of the array is equal to the number of buckets in the table, and the varying column lengths are equal to the bucket sizes. Upon initialization, the hash table hashes every key in the static key-set provided and computes how many keys hash to each row in the ragged array, thereby determining the column lengths (bucket sizes) for each row.

2.11 BioNumPy

BioNumPy is a Python library built on top of NumPy that allows for easy and efficient representation and analysis of biological data [20]. This includes functionality for efficiently and correctly reading a multitude of different file types that are commonly used for storing biological data directly into NumPy arrays, fast encoding from character arrays representing biological sequences into 2-bit encoding for faster processing and better memory efficiency, and *k*mer analysis support.

2.12 KAGE

KAGE [8] ...

2.13 Nucleotide Binary Encoding

DNA nucleotide sequences (described in section 2.1) 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.

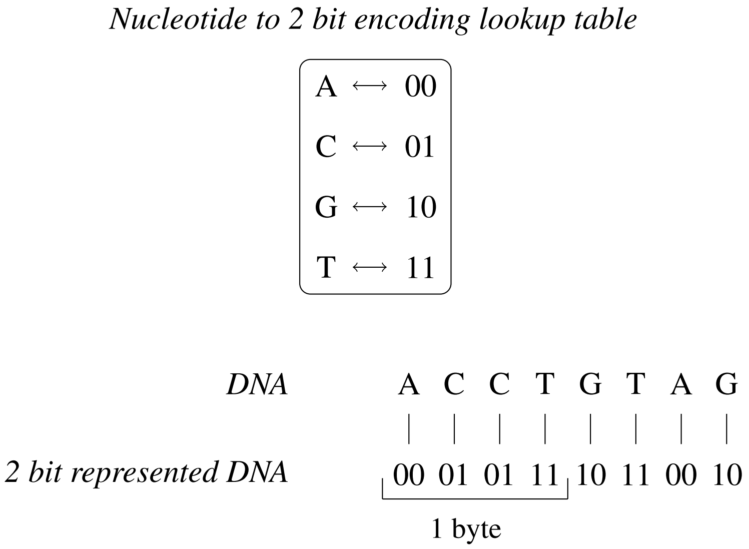


Figure 9: A lookup table showing how nucleotides can be encoded using 2 bits and a DNA nucleotide sequence represented both as plain characters as well as its 2 bit encoded representation. Recall that computers use 8 bits to represent a single nucleotide with a character, whilst the 2 bit encoding only needs 2 bits to represent a nucleotide.

3 Thesis Goal

This thesis has a two main goals. One of the goals is to explore whether state-of-the-art genotyping can be sped up in any significant way by utilizing GPUs. More specifically, this thesis will investigate whether alignment-free genotyping, which presently is significantly faster compared to alignment-based genotyping, can be sped up by utilizing the GPU. In order to investigate this, we will attempt to integrate GPU accelerated functionality into a base alignment-free genotyper, KAGE, which is presently the fastest known genotyper that also yields competitive results. The base genotyper, KAGE, is implemented in Python. This leads to several possible avenues for integrating GPU support, either by low level implementations in C++ using CUDA or using existing Python packages providing GPU accelerated functionality. This leads to the second goal of this thesis - to investigate and experiment with different ways of GPU accelerating exiting Python code (that relies on array-programming libraries such as NumPy), and to discuss the advantages and drawbacks of using the different methods. Finally, GPU accelerated functionality will be integrated into KAGE, resulting in GKAGE (GPU KAGE), and GKAGE will be benchmarked against KAGE to account for any potential speedup.

4 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. We will give an account of how we determined which parts to focus on GPU accelerating and describe a testing strategy that we deployed that allowed us to GPU accelerate existing NumPy code directly in Python to see whether significant runtime speedup was plausible. Then, we will describe how we implemented the GPU accelerated solutions that were introduced into KAGE, resulting in GKAGE.

4.1 Determining which Components to GPU Accelerate

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, whether they would be suitable for the GPU's architecture and 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 bayesian 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 KAGE's total runtime, with *k*mer counting step 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 main focus for GPU acceleration for potential runtime speedup.

4.2 Initial Testing

Before delving into the GPU accelerated methods used to produce the final GKAGE product, we will here describe an initial test we performed in order to assess two things. Firstly, whether we could effectively GPU accelerate existing NumPy solutions using CuPy, all while staying within the comfort of Python, and secondly, whether such an implementation would result in significant speedup over a CPU solution or at least provide insight into how plausible getting significant speedup by utilizing the GPU was. For this, we decided on focusing on the *partial k*mer counting problem, detailed in section 2.5.

As detailed in section 2.10, NumPy Structures is a Python library that enhances the NumPy library by providing additional data structures with NumPy-like behaviour and performance, built on top of NumPy. Additionally in section 2.10, we detailed a subset of NumPy Structures' data structures, one of which was a counter object that efficiently counts occurences of a predefined keys-set in a sample. This counter object had previously been utilized to count a predefined set of *k*mer's frequencies, albeit on the CPU.

Since NumPy Structures' data structures are all heavily reliant upon NumPy's array routines for fast performance, they are designed around utilizing the CPU's vectorization and data parallelism. Large array operations where data parallelism matters are ideal for the GPU architecture, which often can provide orders of magnitude more parallelism this way. In addition, we know that most, if not all of the functionality used from NumPy, will be supported with GPU acceleration by CuPy through an nearly identical interface. Thus, by replacing the NumPy functionality used in NumPy Structures' data structures with CuPy's equivalent GPU accelerated functionality, we can GPU accelerate the necessary data structures in NumPy Structures without having to leave Python. With this strategy we GPU accelerated NumPy Structures' counter object using CuPy.

4.2.1 Implementation

Rather than creating a standalone package version of NumPy Structures with GPU acceleration, we instead opted to add the possibility for GPU acceleration to the already existing package. This added a new self imposed requirement; we did not want CuPy to be a NumPy Structures dependancy since many users may wish to use it simply for the NumPy implementations. Additionally, we still needed a way to redirect NumPy function calls to their equivalent CuPy functions. We fulfilled both of these requirements by exploiting Python's module system.

Consider the following Python package example where our package, *mypackage*, contains two modules, *my_classes* and *my_funcs*, both relying on NumPy for their implementations:

mypackage.my_funcs.py

```
import numpy as np

def some_func_using_numpy():
 return np.zeros(10)
```

mypackage.my_classes.py

```
import numpy as np

class SomeClassUsingNumPy:
 def __init__(self):
  self.data = np.zeros(10)

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

Our package's initialization file imports our function and our class from their respective modules, and all functionality is usable without needing to import CuPy in either module or initialization file. Pay attention to the initialization file's *set_backend* function which takes a library as a parameter and reassigns the np variable in both package modules from the NumPy to the provided library.

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
```

In our own program, where we will import our package, *mypackage*, we can either directly use our package's implementation with NumPy, or we can do as the following example shows and import CuPy and set the backend in the entire package to use CuPy functionality instead of NumPy.

program.py

```
import cupy as cp

import mypackage
mypackage.set_backend(cp)

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

Exploiting Python's module system this way has the benefits of not making CuPy a dependancy for npstructures, and it also allows for gradual GPU support by way of only updating the backend in modules where the existing implementations are ready to be ported as is to CuPy.

4.2.2 Resolving Unsupported or Poorly Performing Functionality

Two issues can arise when utilizing this method of GPU accelerating NumPy code. Firstly, some NumPy solutions may be effective on the CPU's architecture but ineffective on the GPU's, resulting in poor performance. This will often be the case when significant portions of the code needs to be ran in a serial fashion, as opposed to parallel, or when array sizes are too small to mask the overhead of copying data to and from the GPU memory. Secondly, certain NumPy functions are simply not supported by CuPy. In case of the former issue, one might want to create an alternative solution that better utilizes the GPU's strengths, such as its massive parallelism. For the latter issue, there is no other practical option than to create a custom solution that achieves the desired behaviour by using what CuPy functionality is available. We circumvented both of these issues by creating custom implementations where we had the freedom to use the full extend of CuPy's functionality to reproduce the desired behaviour. For classes, this can be achieved by subclassing and overriding methods where we wish to create our new custom implementations. To demonstrate this, we will have to slightly edit our example package, *mypackage*.

Consider if our package's class definition had instead been the following:

mypackage.my_classes.py

```
import numpy as np

class SomeClassUsingNumPy:
  def __init__(self):
    self.data = np.zeros(10)
```

```python

  def pad_with_ones(self):
    arr = self.data
    self.data = np.insert(arr, [0, len(arr)], 1)
```

In the code above, we use NumPy's insert function, which as of April 2023 is not supported by CuPy. To circumvent this issue, we can subclass our *SomeClassUsingNumPy* class and create our own custom implementation of *pad_with_ones*. We will create a new file where we implement our CuPy compatible solution, and our subclass will override the *pad_with_ones* method.

mypackage.cp_my_classes.py

```python
import numpy as np
import cupy as cp

from .my_classes import SomeClassUsingNumPy

class CPSomeClassUsingNumPy(SomeClassUsingNumPy):
  def pad_with_ones(self):
    arr = self.data
    self.data = cp.pad(arr, (1, 1), ’constant’, constant_values=1)
```

In the above example we override the *pad_with_ones* method with our alternative implementation that uses a different CuPy function that is supported by CuPy, the *pad* function. Now, our two different *pad_with_ones* implementations will behave equivalently, although our custom implementation will leverage CuPy and be GPU accelerated.

Finally, we need to make one small change to our package's initialization file so that users of our package will import our CuPy compatible subclass of *SomeClassUsingNumPy* after setting the backend to CuPy:

mypackage.__init__.py

```python
from .my_funcs import some_func_using_numpy
from .my_classes import SomeClassUsingNumPy

# Swaps NumPy with lib (presumably CuPy)
def set_backend(lib):
```

```python
from . import my_funcs
my_funcs.np = lib

from . import my_classes
my_classes.np = lib

# Use CuPy compatible version of SomeClassUsingNumPy
global SomeClassUsingNumPy
from .cp_my_classes import CPSomeClassUsingNumPy
SomeClassUsingNumPy = CPSomeClassUsingNumPy
```

By utilizing this method, we implemented partial GPU support for the NumPy Structures library, including enabling GPU support for the counter object used to count occurences of a predefined key-set in a sample.

4.2.3 Assessment

...

Describe results of this (with graphs) and mention that we have here found a simple way of GPU accelerating existing NumPy code directly in Python without having to dvelve down into C++ and CUDA. Perhaps also mention that the results of this initial testing was not used in the final GKAGE because while it was faster, it was not very fast. Additionally, the implementation of the counter/hashtable created many large array allocations during both initialization and counting, making the memory usage quite extensive in practice, certainly too extensive for a GPU where memory is more scarce.

...

4.3 GPU Accelerating *k*mer Counting

While several *k*mer counting solutions have been developed in previous work, with at least one having support for GPU acceleration [15], most such tools are designed to solve the *full k*mer counting problem, which we described in section 2.5. Additionally in section 2.5, we described how KAGE's *k*mer counting step is a less compute and memory demanding problem which we defined as *partial k*mer counting. As a reminder: we defined *partial k*mer counting as the process of only counting the observed occurences of a predefined set of *k*mers in a sample. Utilizing *k*mer counting tools that are designed to solve the *full k*mer counting problem is futile, and we therefore opted to implement our own GPU accelerated *k*mer counting tool where requirements and design are considered in the context of the *partial* *k*mer counting problem.

4.3.1 Design Requirements and Choices

When designing our new GPU accelerated *k*mer counting solution for KAGE, we had to keep in mind that we were in effect replacing a single component of KAGE, its *k*mer counting solution, with our new GPU accelerated solution. Therefore, a necessary requirement for the new GPU accelerated solution was that it would need to exhibit indistinguishable behaviour to the previous solution in KAGE (except for runtime). In KAGE, the original solution was designed to receive chunks of *k*mers to count in serial in order to restrict memory consumption. A chunk, an array of some parameter N_c number of *k*mers, is counted before a new chunk is generated by reading more data from the genomic data file, 2-bit encoding the sequences and extracting all valid *k*mers (*k*mer hashing). Additionally, our GPU accelerated solution must be functional directly in Python, as KAGE is written in Python.

In order to support counting large chunks of *k*mers at a time, we decided to base our solution on a hash table data structure with support for massively parallel counting of *k*mers as they arrive in large chunks. Using a hash table data structure on the GPU immediately disqualifies certain common paradigms such as open hashing - where each address in the table contains a pointer to a linked list or a tree-type data structure. This is because GPUs, which architecture's are best designed for massive data parallelism performs poorly when having to dereference pointers and performing many strided memory accesses. Our hash table did not need to support changing the key-set after initialization, since we always have a predefined set of *k*mers that we want to count the occurences of. We decided to implement the hash table in C++ using CUDA, Nvidia's programming platform 2.6.2. This would grant us much granular control over the implementation and hardware compared to the CuPy-based method described in section 4.2. Since the hash table would be implemented in C++, we determined to use pybind11 [16] to create Python bindings for our C++ hash table, allowing us to extend its functionality into Python-based KAGE.

4.3.2 Implementation

Unlike the hash table used in the previous method, this hash table uses a simpler open addressing with linear probing scheme for insertions, updates and queries. The hash table data structure is thus a simple structure of arrays, with one array constituting the hash table keys and another the associated values (counts). It is implemented in CUDA as a C++ class and supports the following functionalities:

- key insertion: only performed once upon initialization.
- key counting: for each key, if it exists in the hash table, increment the associated hash table value by one.
- key querying: for each key, if it exists in the hash table, fetch the associated hash table value.

Each of these methods are implemented using custom CUDA kernels, allowing for each operation to happen for a large number of *k*mers in parallel on the GPU. Thus, the insertion, counting and query implementations all expect arrays of *k*mers where a single CUDA thread is assigned to each *k*mer to fulfill the insertion, count update or query.

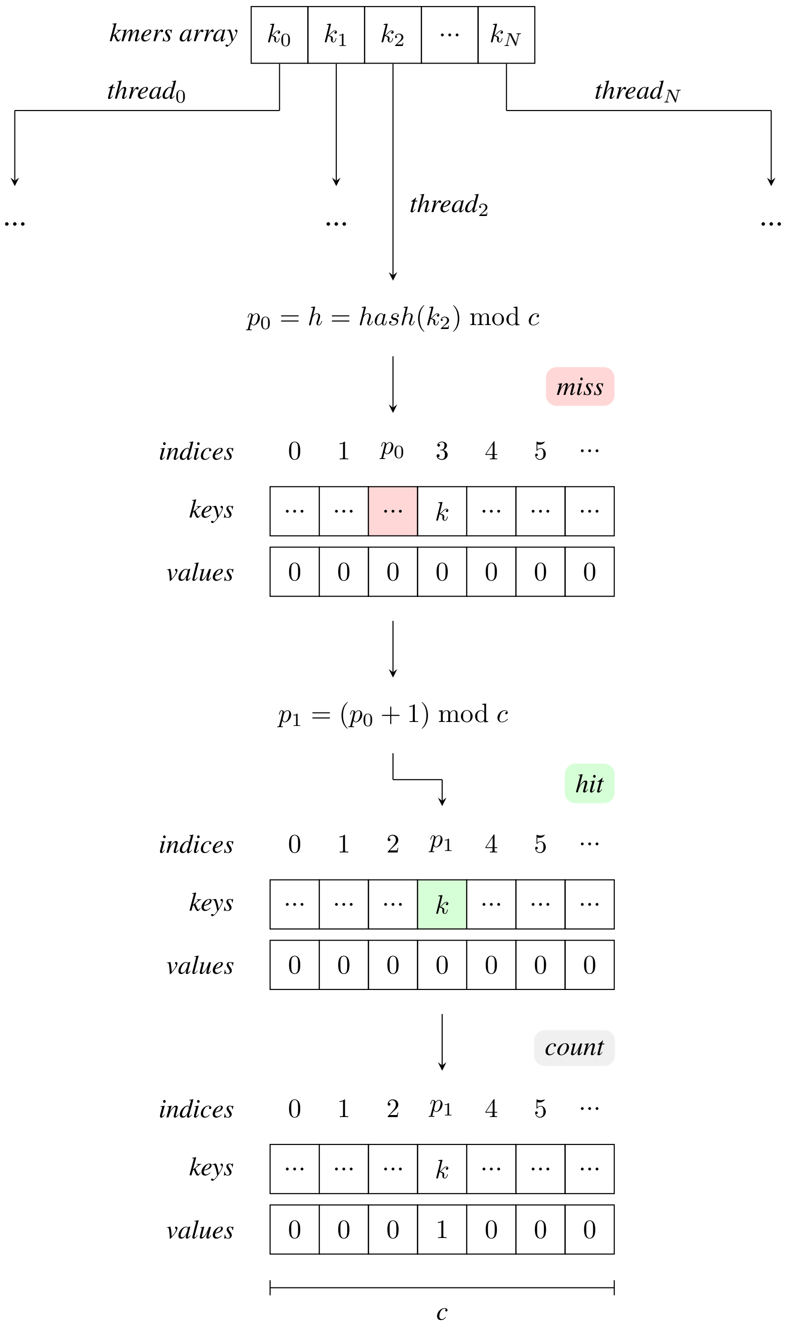


Figure 10: As an array of N 64-bit integer encoded kmers are counted by the hash table, N CUDA threads will launch and each will compute the first probe position p_0 for its assigned kmer k. Then, if the key at slot p_0 does not contain k, it will continue probing by linearly moving up to the next consecutive slot until either an empty key or k observed. If an empty key is observed, the thread terminates without changing any of the hash table's values. If k is observed, the current slot's value is incremented.

4.4 GPU Accelerating *k*mer Hashing

...

5 Results

6 Discussions

6.1 Drawbacks of Graphical Processing Units

While GPUs can be excellent for accelerating many problems in scientific computing, they do come with some notable caveats.

Expensive; power hungry; have experienced significant fluctuation in price and ease of access in the last couple of years (although seemingly mostly because of mining?); can be difficult to make effective solutions for someone inexperienced with the GPU compute models and frameworks; not suitable for every problem, only problems that are possible to rephrase as massively parallel, and it can be difficult to judge whether a problem is suitable before trying

7 Conclusion

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] 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.

[15] Swati C Manekar and Shailesh R Sathe. “A benchmark study of k-mer counting methods for high-throughput sequencing”. In: *GigaScience* 7.12 (Oct. 2018). giy125. ISSN: 2047-217X. DOI: 10.1093/gigascience/giy125. eprint: https://academic.oup.com/gigascience/article-pdf/7/12/giy125/27011542/giy125_reviewer_3_report_(original_submission).pdf. Address <https://doi.org/10.1093/gigascience/giy125>.

[16] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. https://github.com/pybind/pybind11. 2017.

[17] 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>.

[18] 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.

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

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