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 deal 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

A central problem in biology is to effectively uncover and characterize genetic sequence variations in humans. By understanding where and how the human genome varies from individual to individual, we can vastly improve our understanding of how an individual's genetic makeup affects its observable traits - its phenotype. To realize this goal, it is inescapable that we need fast and reliable methods for genotyping - characterizing an individual's genetic makeup - in order to gather data to further explore links between genotypes and phenotypes.

As the price of high-throughput sequencing has steadily become cheaper over the last few decades [1], whole genome sequencing of human genomes has become more accessible than ever before. Today, we can relatively cheaply sequence a whole human genome and expect to receive millions of short polynucleotide reads (often of length $\sim 150$) [2]. From just such reads, where we neither know where the read originates from in the sequence's genome or where it may be erroneous, we want to perform the difficult task of accurately genotyping the individual, and to perform this analysis as quickly as possible.

The traditional and more established methods for genotyping a human individual have involved aligning all sequenced reads to a reference genome to examine where the reads differ from the reference, noting the genotypes supported. Such *alignment-based* methods are computationally expensive and time consuming, with established genotypers such as GATK [3] needing tens of gigabytes of memory and several hours to run [4]. In recent years, new *alignment-free* strategies for genotyping human individuals have emerged. Because of work such as the 1000 Genomes Project [5], we today have access to vast amounts of knowledge about human genetic variation and genotype information. Alignment-free genotyping methods leverage such knowledge to skip the taxing alignment step altogether in favour of genotyping individuals directly based on analysis of *k*mers - short substrings of the read sequences - and previous knowledge of known genetic variation [4, 6]. Recently, a new genotyping tool, KAGE, showed that by deploying an alignment-free method, it is possible to yield competitive genotype prediction accuracies while being an order of magnitude faster than any other known genotyper [4].

In recent years, with the introduction of the *general purpose graphical processing unit* (GPGPU), the *graphical processing unit* (GPU) has become increasingly popular for solving problems that require heavy amounts of compute, with many such problems existing in the space of scientific computing. Common for all traditional genotyping tools is that they are implemented to run on the *central processing unit* (CPU). This likely stems from the fact that

memory usage commonly reaches tens, and sometimes hundreds of gigabytes when running these tools [4], and that not all problems are fit to run on the GPU architecture. However, the currently fastest known genotyping tool, KAGE, requires significantly less memory compared to its competitors. Additionally, KAGE is implemented in Python, and performs a significant amount of large array operations - a type of operation well suited for the GPU architecture - using the Python library NumPy [7]. Because of this, KAGE seemingly stood to benefit from GPU acceleration.

In this thesis, we will explore whether we can speed up (alignment-free) genotyping in any significant way by utilizing the GPU. We will start with KAGE as a base genotyper, and explore possible avenues for implementing GPU acceleration in KAGE to assess whether it results in significant speedup. Since KAGE is implemented in Python and consists of several steps that can in effect be considered independent, several possible avenues for GPU acceleration are possible. We will explore several of them to assess what options developers may have to GPU accelerate existing work in Python, and evaluate the advantages and drawbacks of each.

# 2 Thesis Goal

This thesis has 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 existing 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.

# 3 Background

## 3.1 Biology

### 3.1.1 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 [8]. 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 [9, 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 [9, 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 [10].

The term *genome* can be used to refer to the complete genetic material of an organism. In prac-

tice, however, the genome of an organism often simply refers to the complete DNA nucleotide sequence of one set of chromosomes for that organism [9, 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 [3].

### 3.1.2 High-Throughput DNA sequencing

*High-throughput sequencing* (HTS), also known as *next-generation sequencing* (NGS), refers to an assortment of recently developed technologies that parallelize 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* [2], all the way up to 15-20 thousand bases using *Pacific Biosciences* (PacBio) technologies, referred to as *long reads* [12]. 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.

example.fa

```
>read 0
ACGTATGCGGCGGGGCGCGATTATTCGTTGCGTATGC
>read 1
ACACGTCGTGCGTAGCGTGTCAGTCACAGTAAACAAA
>read 2
CGTTGCCATCAACGGCTGTGCACGATTGGGGGCGCGC
...
```

Figure 2: An illustration of how sequenced DNA reads are stored as plain text in a FASTA (.fa) file. Before each read, a header file beginning with the character ">" may provide information about the read.

### 3.1.3 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. The traditional way of performing variant calling is to use an alignment-based method, which can abstractly be described as the following three steps: 1) sequence the genome of interest to get DNA reads (described in section 3.1.2), 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 [13]. A popular alignment-based tool for performing variant calling is GATK [3].

Variants found in genomes of individuals of the same species can take many different forms. Three common types of variants are:

- *Single nucleotide polymorphism* (SNP) is a variant in which only a single nucleotide differs in the sequenced genome when compared to a reference genome at a location of interest.
- *Indel* refers to one of two types of variants: *insertion* - a sequence of nucleotides, not present in the reference genome, has been introduced in the sequenced genome, and *deletion* - a sequence of nucleotides, present in the reference genome, are missing in the sequenced genome.
- *Structural variant* is a large-scale genetic variation in which several alterations in the structure have occurred, typically affecting more than 1000 bases.

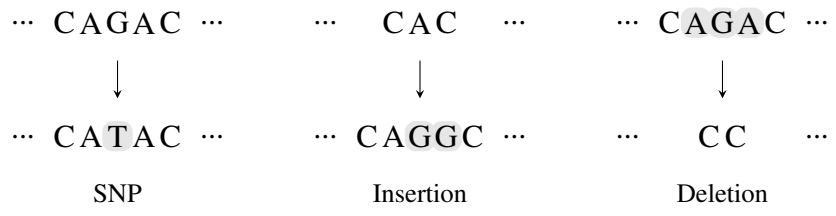


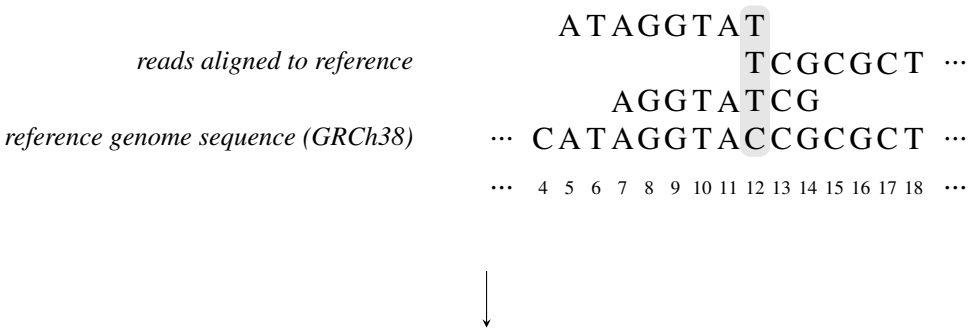

Figure 3: An illustration demonstrating how three types of variants can manifest when aligning sequenced reads to a reference genome. From left to right: a single nucleotide polymorphism (SNP), an insertion (indel), and a deletion (indel).

The software tool KAGE, which is described in section 3.6, focuses on genotyping SNPs and indels.

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 [14]:

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 variant.
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 4: 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.

### 3.1.4 Genotypes and Genotyping

Recall that a genetic variant refers to a distinct variation at a particular location along an individual's genome when compared to a reference genome sequence. For humans, who have two of each chromosome, we may have the same, or two different variations present at a particular location in the genome. The term *genotype* refers to the set of variants an individual carries at such a location. *Genotyping* an individual refers to the process of determining which genotypes an individual carries. For humans, this would entail determining the set of variants present at each variant site. 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.

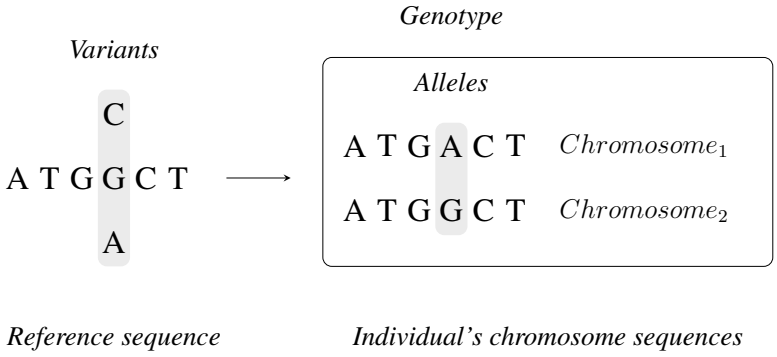



Figure 5: In humans, where there are two chromosomes, a genotype constitutes a set of two alleles, one in each chromosome at the variant location. Along the reference sequence on the left, several possible variants may be known to occur at a specific location. After examining the sequence of an individual, 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 [3]. However, given how many reads one have come to expect from high-throughput sequencing today [3.1.2] 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 [4, 6, 5]. One such alignment-free genotyper, KAGE, has recently shown that it can genotype a human individual more than 10 times faster than any other known genotyper tool, while still providing competitive accuracy scores [4].

## 3.2 Nucleotide Binary Encoding

DNA nucleotide sequences (described in section 3.1.1) 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, often referred to as 2-bit encoding. 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.

A $\longleftrightarrow$ 00
C $\longleftrightarrow$ 01
G $\longleftrightarrow$ 10
T $\longleftrightarrow$ 11

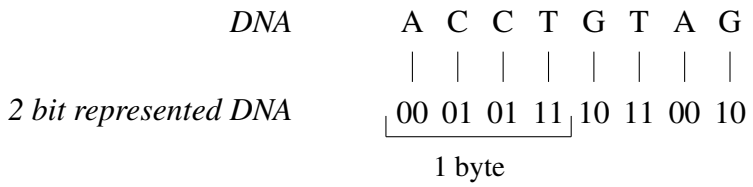



Figure 6: A lookup table showing how nucleotides can be encoded using 2 bits and a DNA nucleotide sequence represented both as plain characters as well as its 2 bit encoded representation. Recall that computers use 8 bits to represent a single nucleotide with a character, whilst the 2 bit encoding only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

## 3.3 *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 3.2, 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 3.6. 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 occurs in those sequences. This problem is commonly referred to as $k$mer counting.

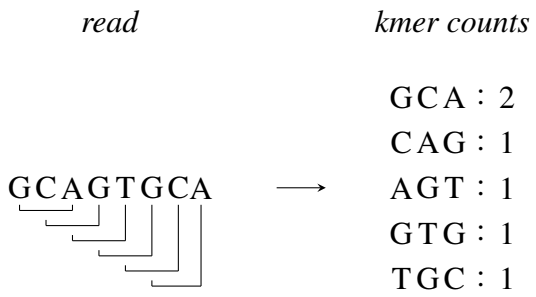



Figure 7: *Full* $k$mer counting where we count the observed frequency of every valid $k$mer in our read set.

The genotyping software tool KAGE, detailed in section 3.6, 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.

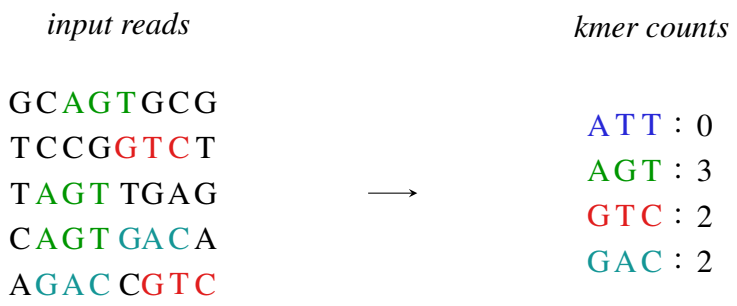



Figure 8: *Partial* *k*mer counting where we only count the observed frequencies of *k*mers present in a predefined set. In this example, our set of predefined *k*mers is {ATT, AGT, GTC, GAC}. During counting, *k*mers not present in this set are skipped.

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 occurrence as *full* *k*mer counting, and to the latter process where we only count the occurrences 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 [15], having support for GPU acceleration [16], these tools are designed to solve the *full* *k*mer counting problem.

## 3.4 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). The concept of the GPGPU is to use a GPU to accelerate computations in other domains where CPUs have traditionally been used. Fields such as artificial intelligence and the broader scientific computing community have enjoyed great utility from GPUs, using them to accelerate embarrassingly parallel problems, e.g., matrix operations. Despite being comparable 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.

While several distinct brands of GPUs exist, the work presented in this thesis only leverages GPUs produced by Nvidia, one of the leading accelerated computing manufacturers for scientific computing today [17].

### 3.4.1 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.

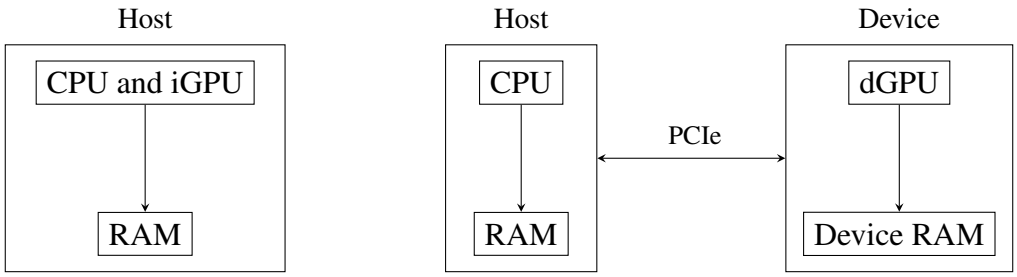



Figure 9: **Left**: A computer setup with a CPU and an iGPU sharing the same die and the same physical RAM. **Right**: A computer setup where a dGPU is connected over PCIe and the dGPU has its own physical RAM, adding the overhead of copying data both to and from the host computer when utilizing the 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 forth 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.

### 3.4.2 CUDA

CUDA [18] is Nvidia's general purpose parallel computing platform and programming model that allows software engineers to leverage the parallel compute engines in Nvidia GPUs. Although the CUDA software environment can be imported in several supported programming

languages, development of GPU accelerated programs have traditionally been done in C++ with CUDA functionality imported through header files [18].

The CUDA programming model is designed to be utilized to solve parallel problems where the GPU is used. To programmers, the programming model is made up of a hierarchy of three units of different granularity: *threads*, *blocks*, and *grids*.

#### 3.4.2.1 Threads, Blocks and Grids

CUDA threads are the most granular units of parallelism in the CUDA programming model. Thread blocks are collections of threads that can be either one-, two- or three-dimensional. Grids are at the top of the hierarchy, as collections of thread blocks. Both thread blocks and the grids can either be one-, two- or three-dimensional. Thus, their dimensions are referred to as $x$ for width, $y$ for height and $z$ for depth. In the CUDA model, each individual thread can be distinctly identified by its index in a thread block, and the thread block's index in a grid. When developing parallel programs, each thread has direct access to its (up to) three-dimensional index inside its thread block, and its thread block's (up to) three-dimensional index in its grid. If a thread resides in a three-dimensional thread block of dimensions $(Dx_b, Dy_b, Dz_b)$ at index $(x_b, y_b, z_b)$, its unique index within said block can be computed using:

$$i_b = (x_b + y_b Dx_b + z_b Dx_b Dy_b) \tag{1}$$

Then, we can include the grid as well to compute the system-wide unique index for the thread. If the thread's thread block resides in a grid of dimensions $(Dx_g, Dy_g, Dz_g)$ at index $(x_g, y_g, z_g)$, its unique system-wide index can be computed using:

$$i = (x_g + y_g Dx_g + z_g Dx_g Dy_g)(Dx_b Dy_b Dz_b) + i_b \tag{2}$$

This thread-hierarchy provides a natural way to perform computations over elements in i.g. vectors, matrices or tensors. All threads in a thread block must reside on the same *stream multiprocessor core* in the GPU. Therefore, there is a size limit on how large thread blocks can be. The two types of Nvidia GPUs used in this thesis, the Nvidia Tesla V100 and the Nvidia GTX 1660 SUPER, have limits of 2048 and 1024 threads per thread block respectively.

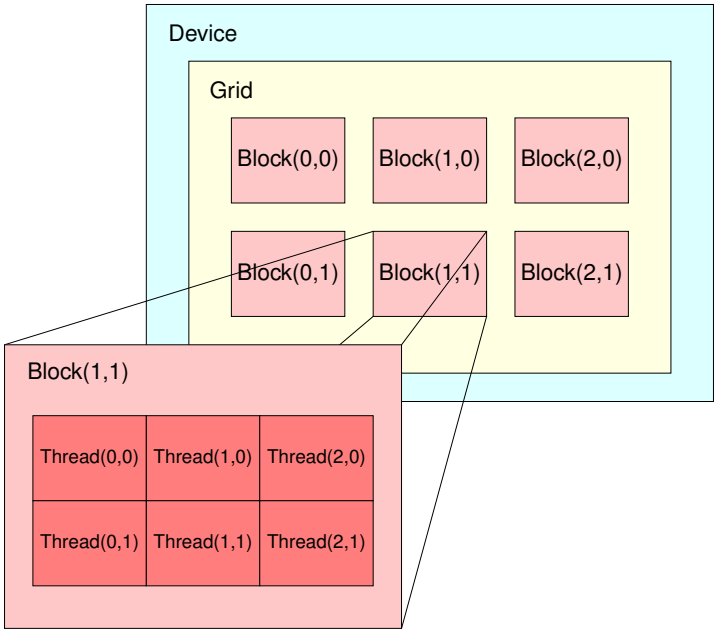



Figure 10: An overview of the CUDA programming model. A single two-dimensional grid contains two-dimensional thread blocks.

#### 3.4.2.2 Kernels

When writing parallel programs for the GPU using CUDA, *kernel* functions define how the GPU should solve problems in a parallel fashion by using the thread-hierarchy. Kernels are functions that, when called, launch once for each thread in the thread-hierarchy configurations, all in parallel. Thus, the code that resides in the kernel definition is executed in parallel by every distinct CUDA thread. Inside the kernel, each CUDA thread has access to its unique index within its thread block as well as unique thread block index within its grid. The thread blocks' dimensions and number of threads are typically determined by the problem and limited by the max number of threads per streaming multiprocessor core on the system. For the grid, the size is typically determined by the size of the problem, and computed as a result of how many thread blocks must be launched to fully complete the necessary computations.

Below is a simple example program where a kernel is implemented to increment every integer in a vector by one.

main.cu

```
// Kernel definition
__global__ void increment_kernel(int *array, size_t N) {
  int i = (threadIdx.x * blockDim.x) + threadIdx.x;
  array[i] = array[i] + 1;
}

int main() {
  size_t N = 1048576;
  // Allocate and initialize array on the GPU
  int *array = ...

  // Determine grid- and block-dimensions and then launch kernels
  dim3 block_dim(1, 1, 1028);
  dim3 grid_dim(1, 1, N / 1028);
  increment_kernel<<<grid_dim, block_dim>>>(array, N)
  ...
}
```

Figure 11: A simple example program showing how the CUDA thread-hierarchy is utilized in a kernel to implement GPU acceleration for incrementing every value in an integer array.

The example program above is implemented in C++. Certain keywords used are CUDA specific, and are not supported without including the necessary CUDA headers beforehand. `__global__` is used to declare that a function is a kernel. The number of CUDA threads launched to run a kernel is specified in the `<<<...>>>` part of the kernel call.

#### 3.4.2.3 Memory Hierarchy

Each CUDA thread has access to several distinct memory areas. Every distinct CUDA thread has its own private local memory. Here, the thread will store small amounts of data such as registers. If a thread needs more memory than it has available through its private local memory, it will turn to the GPU's global RAM memory as a fallback, which significantly hurts performance. Each thread block has a shared memory region accessible by threads in said block. This memory region shares its lifetime with its associated block. The GPU's global RAM is the largest memory region in the GPU, typically ranging from a few to several gigabytes. The global memory is several orders of magnitude slower compared to the more local memory regions, so effective memory access patterns are paramount for performance.

#### 3.4.2.4 Programming and Considerations

The GPU is described by Nvidia to be a *Single Instruction Multiple Threads* (SIMT) machine [18]. Although CUDA threads are described as threads, 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. Therefore, it is important to keep in mind that code branching may damage performance. Furthermore, because of the massive amounts of transistors dedicated to processing in the GPU, the bottlenecks in parallel programs on the GPU is typically memory bandwidth. Particularly global memory requests are expensive and should be avoided when possible. Ideally, consecutive CUDA threads should access consecutive data from the GPU's global RAM memory. This allows for the GPU to make larger and fewer memory requests, resulting in significantly better memory bandwidth and overall performance.

## 3.5 Implementation Tools and Libraries

### 3.5.1 C and C++

C and C++ are compiled general purpose programming languages. C++ is a superset of C, and can in effect be considered to be C with classes among some other high-level functionality. C and C++ are popular choices for implementing optimized and performant code. They offer granular control over hardware and memory, where data such as arrays must be both allocated and de-allocated manually.

### 3.5.2 Python

Python is an interpreted high-level general purpose programming language that has gained significant traction, including in the scientific computing community, in recent years [19, 20]. Since it is an interpreted language, Python in and of itself is not performant, and can in many instances be measured to be several orders of magnitude slower than similar implementations in performant compiled languages such as C and C++. The Python interpreter is written in C, and thus, the Python programming language can naturally be extended with additional C and C++ code. Because of its high-level and interpreted nature, it is well suited for quick prototyping of software. However, because of its ability to be extended with additional C and C++ code, many packages exist today that supplement Python with high-performance libraries, making Python a serious language candidate for scientific computing software today, albeit through calling upon optimized and performant C and C++ code.

### 3.5.3 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 [7]. 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 parallelize 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.

### 3.5.4 CuPy

CuPy is GPU accelerated NumPy [7] and SciPy [21] compatible array library that, much like NumPy [7], 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.

CuPy also offers some access to CUDA functionality such as CUDA streams used to copy memory to and from the GPU's memory while simultaneously processing data, and device synchronization to halt the CPU until a process started on the GPU has completed. Additionally CuPy provides support for creating custom kernels that can operate on GPU allocated arrays, directly in Python. These kernels are then JIT (just-in-time) compiled when the program first encounters the kernel. Thus, CuPy provides a useful module where GPU accelerated implementations can be made directly with a NumPy-like array interface, while also supporting more granular custom kernels that can operate on the data in these arrays, all directly inside Python.

### 3.5.5 npstructures

npstructures (NumPy Structures) is a Python package built on top of NumPy that provides data structures with NumPy-like features to augment the NumPy library [22]. This is achieved by building these new data structures using NumPy's underlying multi-dimensional array object and fast array routines.

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

#### 3.5.5.1 Ragged Array

A central feature in npstructures 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 12: A simple illustration of how npstructures' data structures can be used directly in Python as drop-in augmentations to the NumPy library.

#### 3.5.5.2 Hash Table and Counter

npstructures 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 occurrences 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.

### 3.5.6 BioNumPy

BioNumPy is a Python library built on top of NumPy that allows for easy and efficient representation and analysis of biological data [23]. 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.

### 3.5.7 pybind11

pybind11 [24] 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 seamlessly 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 revolves around 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 was, in these cases, crucial in order to create the bindings from these C++ implementations, making them usable directly in Python.

## 3.6 KAGE

KAGE [4] is an alignment-free genotyping tool for SNPs and short indels. As opposed to alignment-based genotyping tools, KAGE relies on an alignment-free method where genotype-probabilities are computed using a statistical model. These genotype-probabilities are supported by *k*mer frequencies found in the input reads and previous knowledge of genetic variation and genotype information from thousands of individuals, produced by studies such as the 1000 Genomes Project [5]. KAGE recently showed (2022) that its accuracy was on par with

the best existing alignment-free genotyping tools while being an order of magnitude faster [4].

KAGE is implemented in Python and relies heavily on the NumPy library for performance. Its genotyping pipeline is split into two distinct programs: *kmer_mapper*, solving the *partial* *k*mer counting problem of counting *k*mer frequencies for a predefined set of *k*mers in the input reads, and *kage*, which finally computes the genotype-probabilities using a statistical model.

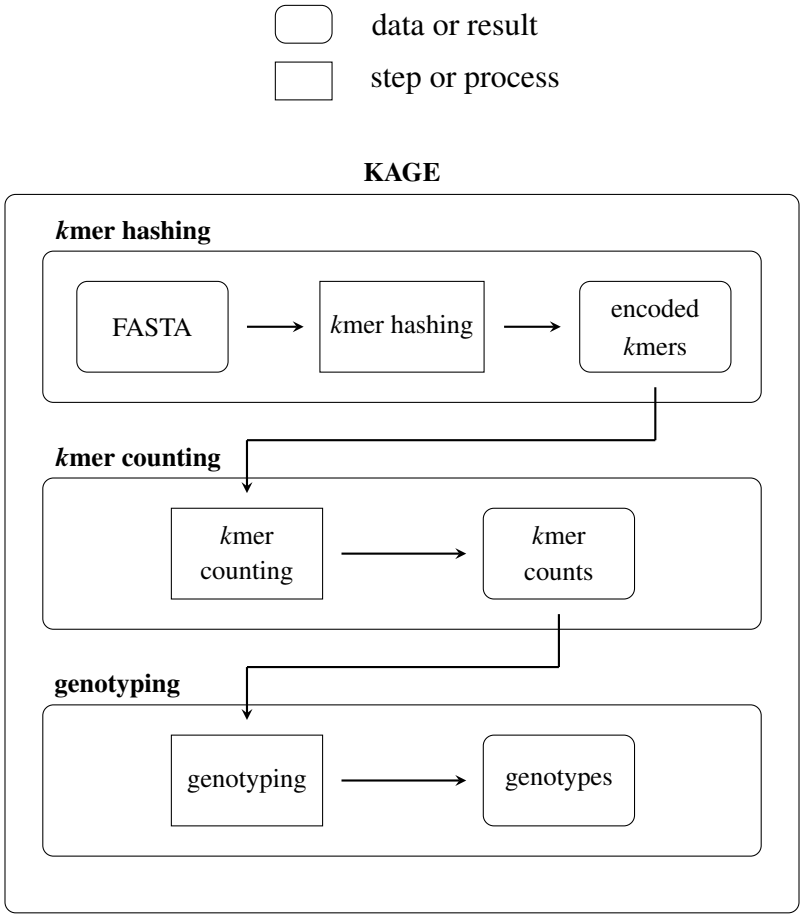



Figure 13: A simplified illustration of how the KAGE genotyping pipeline works. The two top most rows, going from an input FASTA file to *k*mer counts, are implemented as an individual piece of software. Three important parts of the pipeline are shown as distinct processes in the illustration: 1) ***k*mer hashing**, which refers to the process of 2-bit encoding input reads from the fasta file and extracting - *hashing* - all valid *k*mers from those reads, 2) ***k*mer counting**, which refers to the process of *partial* *k*mer counting - counting the observed frequencies of a predefined set of *k*mers in the input reads, and 3) **genotyping**, which refers to the process of computing the final genotype-probabilities.

# 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. For each GPU acceleration method, we include an assessment showing the effects of GPU acceleration and how they compare to previous CPU runtimes.

## 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 KAGE genotyping pipeline 13 is split into two distinct processes (or programs): 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 3.3.

As detailed in section 3.5.5, npstructures 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 3.5.5, we detailed a subset of npstructures' data structures, one of which was a counter object that efficiently counts occurrences 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 npstructures' 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 a nearly identical interface. Thus, by replacing the NumPy functionality used in npstructures' data structures with CuPy's equivalent GPU accelerated functionality, we can GPU accelerate the necessary data structures in npstructures without having to leave Python. With this strategy we GPU accelerated npstructures' counter object using CuPy.

### 4.2.1 Using CuPy as a Drop-In Replacement for NumPy

Rather than creating a standalone package version of npstructures 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 npstructures dependency 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

```
1 import numpy as np
2
```

```
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 dependency 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 the 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 extent 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)
```

```
6
7   def pad_with_ones(self):
8     arr = self.data
9     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

```
1 import numpy as np
2 import cupy as cp
3
4 from .my_classes import SomeClassUsingNumPy
5
6 class CPSomeClassUsingNumPy(SomeClassUsingNumPy):
7   def pad_with_ones(self):
8     arr = self.data
9     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

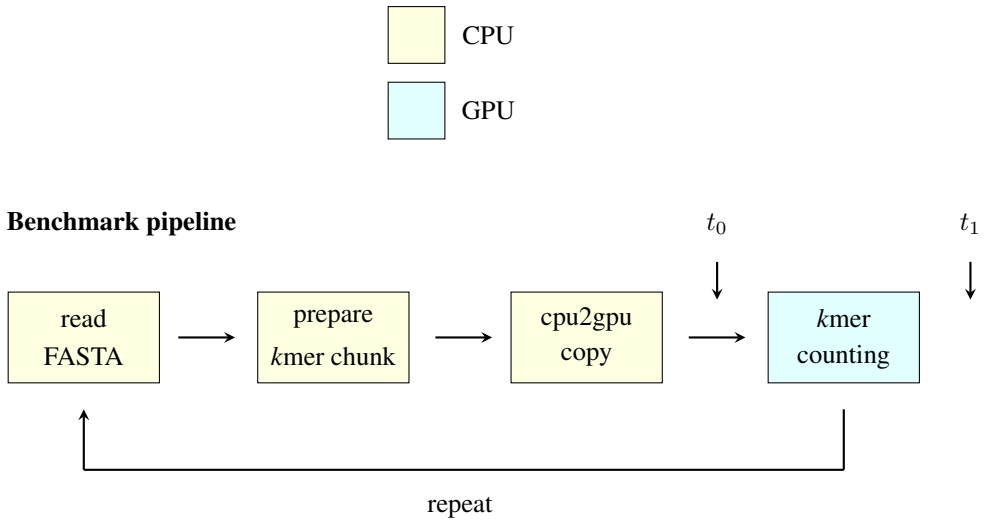
```
1 from .my_funcs import some_func_using_numpy
2 from .my_classes import SomeClassUsingNumPy
3
4 # Swaps NumPy with lib (presumably CuPy)
5 def set_backend(lib):
```

```
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 npstructures library, including enabling GPU support for the counter object used to count occurrences of a predefined key-set in a sample.

### 4.2.3 Assessment

To assess the effect GPU accelerating npstructures by using CuPy as a drop-in replacement for NumPy, we benchmarked the GPU accelerated npstructures counter object by counting the occurrences of a predefined set of 50 million unique *k*mers in a set of 20 million DNA reads, each of length 150. This benchmark used BioNumPy to read the FASTA file in chunks of 10 million bytes, 2-bit encode the reads and hash the valid *k*mers before counting. The time spent preparing each chunk was not included in the final measured runtime, since we were only interested in seeing the effects of GPU accelerating npstructures' counter object. It is important to note that while this benchmark provided a good assessment of the speedup gained from GPU accelerating our counter, it omits the overhead of transferring each chunk of *k*mers to the GPU in order to be counted by our GPU accelerated counter. We justified the omission of this overhead because of two reasons. Firstly, during our initial testing we were only interested in assessing the runtime differences during processing on the GPU versus the CPU when using CuPy as a drop-in replacement for NumPy. Secondly, we only GPU accelerated one component of a much larger pipeline, and in such a case, the true effects of GPU acceleration may be overshadowed. Our intention was to GPU accelerate several consecutive components in a way where only a single copy step would need to be performed before several consecutive processing steps could benefit from GPU acceleration, minimizing the effects of the CPU to GPU copy overhead.




$$\textit{counting time} = \sum_{i=1}^{\textit{num chunks}} t_1^i - t_0^i$$

Figure 14: The pipeline used to benchmark the CuPy-based against the NumPy-based npstructures counter object. Only the calls made to the counter object to count chunks of *k*mers are timed, and these times are finally summed to get the total runtime spent on counting *k*mers. The difference between the two runtimes are then used to infer the effects of GPU accelerating npstructures' counter object using our initial testing method.

We ran the *k*mer counting pipeline and measured the time spent counting *k*mers for both the NumPy-based CPU version of the counter and the CuPy-based GPU version. The total time spent counting *k*mers was measured by timing each call to the counter object and summing these runtimes in order to get the total counting time. For the GPU version, we made sure to synchronize the GPU to the CPU between each call, since kernel calls made from the CPU to the GPU are asynchronous otherwise. This benchmark yielded the following results:

| **Method** | **Counting time (seconds)** |
|---|---|
| NumPy | 234.7 |
| CuPy | 24.31 |

Table 1: The total time spent counting *k*mers using npstructures' counter object, one with NumPy as the backend array-library, running on the CPU using one thread, and the other using CuPy as the backend, running on the GPU.

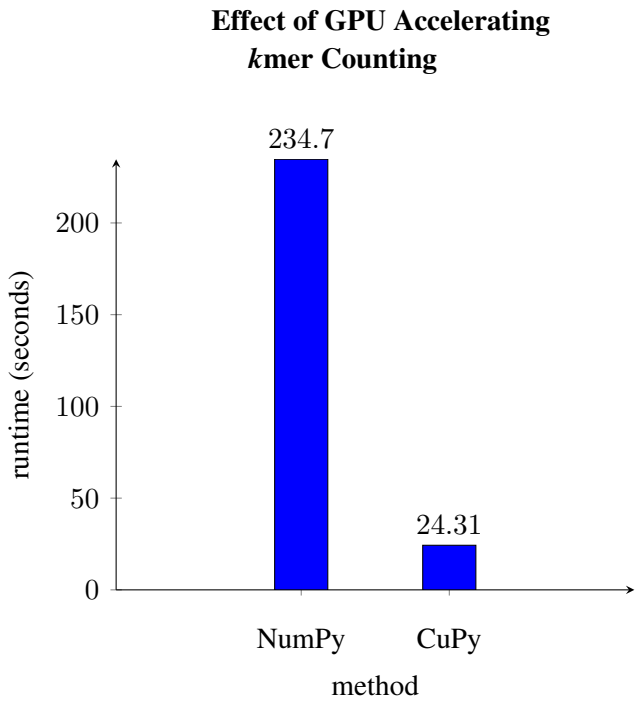



Figure 15: Elapsed time counting the occurrences of 50 million unique *k*mers in a set of 20 million reads, each of 150 bases. Both methods use BioNumPy to read, 2-bit encode and hash chunks of *k*mers, reading 10 million byte chunks from the FASTA at a time. Both methods also use npstructures' NumPy-based counter object to count the *k*mer frequencies, although the CuPy-based version replaces NumPy with CuPy to achieve GPU acceleration.

As can be seen in table 1, simply using CuPy as a drop-in replacement for NumPy in npstructures to GPU accelerate its counter object yielded a nearly 10X speedup in counting efficiency, without having to otherwise change the implementation or even leave Python.

## 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 [16], most such tools are designed to solve the *full* *k*mer counting problem, which we described in section 3.3. Additionally in section 3.3, 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 occurrences of a predefined set of *k*mers in a sample. Repurposing *k*mer counting tools that are designed to solve the *full* *k*mer counting problem is futile if the goal is speedup. Thus, we opted to implement our own GPU accelerated *k*mer counting tool where its design is considered in the context of the *partial* *k*mer counting problem.

Although we had success in GPU accelerating a hash table in npstructures using CuPy, we

decided to also attempt to implement a hash table directly in C++ using CUDA, Nvidia's programming platform. This would allow for a more granular implementation as we would no longer be constrained to a solution originally designed for NumPy array functions.

### 4.3.1 GPU Hash Table Implemented using CUDA

The hash table's interface needed to support three main operations: 1) **insert** - insert each *k*mer found in an input array of *k*mers (although only once upon initialization), 2) **count** - increment the value associated with each *k*mer found in an input array of *k*mers, and 3) **query** - fetch the values associated with each *k*mer found in an input array of *k*mers.

Common for all three operations mentioned above is that they need to adhere to an addressing and probing scheme. Since our hash table resides on the GPU, certain common paradigms such as open hashing - where each address in the table contains a pointer to a linked-list or tree-type data structure for placing values - are immediately disqualified. This is because GPUs, which architectures are designed for massive data parallelism, perform poorly when needing to dereference pointers and make many strided memory accesses. Thus, we went with a simple scheme using open addressing and linear probing.

Two arrays of equal size make up the data structure where one array stores the table's keys (*k*mers) and the other array stores its values (counts). The keys array is an array of 64-bit unsigned integers. The choice of using 64 bits for the keys, as opposed to *i.e.* 32, was to accommodate for larger *k*mer values since 32 bits would only allow for *k* up to $(32/2)-1 = 15$. Using 64-bit keys increases our maximum *k* size to 31, which is the default value for KAGE. This choice also reveals a limitation of our hash table, which is that it can not support *k*mers where $k > 31$ in its current form. The reason why 64 bits does not allow for *k* up to 32, considering a 2-bit encoded 64-mer can be stored in a single 64-bit integer, is that the max 64-bit integer value is used as an indicator that a slot is empty in the hash table. The values array is an array of 32-bit unsigned integers. While 16-bit unsigned integers would suffice for our purpose of counting *k*mers, the choice of using 32-bit unsigned integers instead was made because the CUDA framework does not have a readily usable implementation of the atomicAdd function for 16-bit integers.

The probing scheme we used - linear probing - describes how we solve collisions in the hash table. This scheme is shared for all three operations supported by the hash table: insert, count and query. A murmur hash [25] is used to hash and compute the first search index for a *k*mer. The initial probe index $p_0$ and every consecutive probe index $p_i$ for a *k*mer $k$ can be found

using:

$$p_0 = hash(k) \bmod c \qquad\qquad p_i = (p_{i-1} + 1) \bmod c \tag{3}$$

where $hash$ is the murmur hash function and $c$ is the capacity of the hash table (the length of the arrays).

For example, in a hash table of capacity $c$, we query a *k*mer $k$ using the following algorithm:

```
1  input: uint64[] keys, uint32[] values, int c, uint64 k
2  begin
3    h ← murmur(k) mod c
4    while true do
5      if keys[h] = k then
6        return values[h]
7      else if keys[h] = empty then
8        return 0
9      else
10       h ← (h + 1) mod c
11     end if
12   end while
13 end
```

Figure 16: The algorithm used for querying keys (*k*mers) in the parallel GPU hash table implemented in C++ using CUDA. When querying an array of $N$ *k*mers, $N$ CUDA threads are launched and each thread is assigned a *k*mer. The threads then perform this algorithm in a parallel fashion to query many *k*mers in parallel. The probing scheme used in this algorithm is the same used in both insertions and updating of values.

Using a a murmur hash function when computing the initial probe position in the hash table results in more uniform placements as opposed to if we only use the *k*mer's integer value instead. This in turn results in less clustering when inserting *k*mers, meaning we need to deal with fewer collisions.

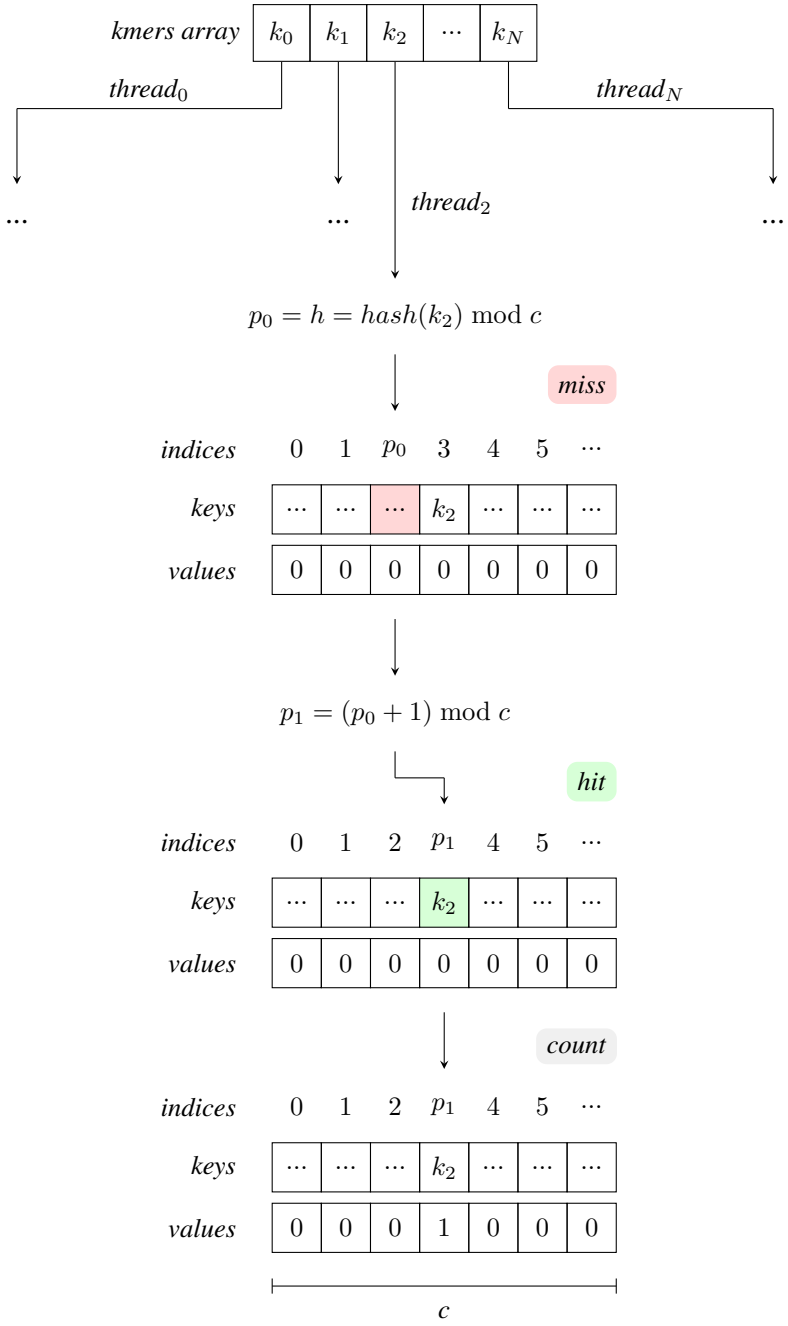



Figure 17: As an array of $N$ 64-bit integer encoded kmers are counted by the hash table, $N$ CUDA threads will launch and each will compute the first probe position $p_0$ for its assigned kmer $k$. Then, if the key at slot $p_0$ does not contain $k$, it will continue probing by linearly moving up to the next consecutive slot until either an empty key or $k$ is 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.

#### 4.3.1.1 Integration to Python

In order to integrate the hash table implemented in C++ using CUDA to Python so that it would be compatible with KAGE, we used pybind11 (introduced in section 3.5.7) to create Python bindings for the C++ class and then wrap the implementation in a Python class. pybind11 provides easy-to-use macros for this, allowing us to add bindings by simply creating a bindings

C++ file and then compiling a module where our C++ functionality is contained.

bindings.cpp

```cpp
#include <pybind11/pybind11.h>
namespace py = pybind11;

int my_function(int a, int b) {
  return a + b;
}

class MyClass {
public:
  MyClass(int number) : number(number) {}
  int get_number() const { return number; }
private:
  int number;
};

// Use pybind11's PYBIND11_MODULE macro to create Python bindings
    for our simple function and class
PYBIND11_MODULE(my_module, m) {
  m.doc() = "Documentation for my module";

  m.def("my_function", &my_function);

  py::class_<MyClass>(m, "MyClass")
    .def(py::init<int>())
    .def("get_number", &MyClass::get_number);
}
```

Figure 18: An illustration of how pybind11 can be used to create Python bindings for C++ functions and classes using a simple C++ macro: PYBIND11_MODULE. The C++ project can then be compiled to produce the module that can be imported directly into Python. pybind11 takes care of translating common data types automatically, including some data structures such as tuples and lists. In addition, pybind11 has specific support for NumPy, making the usage of NumPy arrays seamless.

The produced module is finally wrapped around a Python class that takes care of any transitional details between usage and the C++ interface, such as determining whether to call upon a version of a function that expects NumPy arrays with data allocated in the host's RAM or a CuPy array with data allocated in the GPU's RAM.

The final implementation of this CUDA accelerated hash table can be found and installed at https://github.com/kage-genotyper/cucounter.

### 4.3.2 Assessment

In order to assess this new CUDA hash table implementation, we ran the benchmark described in section 4.2.3, but using the CUDA hash table with Python bindings for *k*mer counting instead of npstructures' counter object. The benchmark yielded the following results:

| **Method** | **Counting time (seconds)** |
|---|---|
| NumPy | 234.7 |
| CuPy | 24.31 |
| CUDA | 2.94 |

Table 2: The total time spent counting *k*mers using the CUDA hash table implemented in C++ with Python bindings. NumPy and CuPy refers to the previously found runtimes using npstructures' counter object with NumPy- and CuPy-backends respectively.

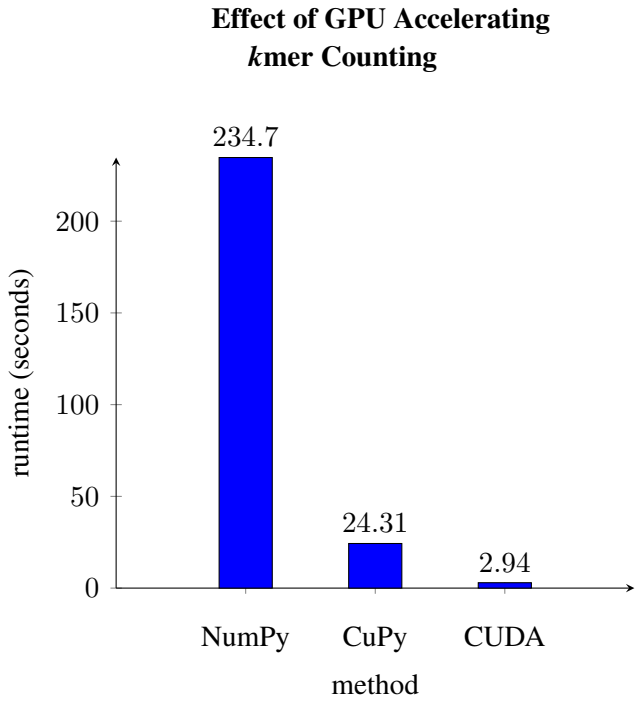



Figure 19: Elapsed time counting the occurrences of 50 million unique *k*mers in a set of 20 million reads, each of 150 bases.

As can be seen in table 2, the CUDA hash table implemented in C++ with Python bindings provided significantly better counting efficiency compared to the CuPy-based npstructures counter, with nearly 80X speedup compared to the original NumPy-based counter and more than 8X speedup compared to the CuPy-based counter.

### 4.3.3 Re-Implementing the Hash Table Directly in Python

While the CUDA hash table we implemented in the previous section yielded significant speedup when *k*mer counting and much better results than the GPU accelerated version of npstructures' hash table, it required us to delve into C++, using CUDA's programming framework and leaving the comforts of Python behind in order to implement. We therefore explored another possible avenue for implementing such a hash table, this time directly in Python. CuPy offers more than just a GPU accelerated subset of NumPy's array interface. Additionally, CuPy allows for custom kernels written directly in Python, to be *jit* (just-in-time) compiled. By exploiting this, we were able to re-implement our parallel hash table directly in Python using CuPy's jit functionality. This implementation can be found online at https://github.com/jorgenwh/cupycounter.

### 4.3.4 Assessment

Again, we used the same benchmark used to assess the CuPy-based npstructures counter and the CUDA hash table (section 4.2.3), to also assess the CuPy JIT (just-in-time) compiled implementation where we wrote custom kernels to get near-identical behaviour and performance as the CUDA hash table, all directly in Python. The benchmark yielded the following results:

| **Method** | **Counting time (seconds)** |
|---|---|
| NumPy | 234.7 |
| CuPy | 24.31 |
| CUDA | 2.94 |
| CuPy JIT | 2.99 |

Table 3: The total time spent counting *k*mers using the just-in-time compiled kernels written directly in Python using CuPy's custom kernel support. NumPy, CuPy and CUDA refers to the previous benchmark results from section 4.2.3 and 4.3.2.

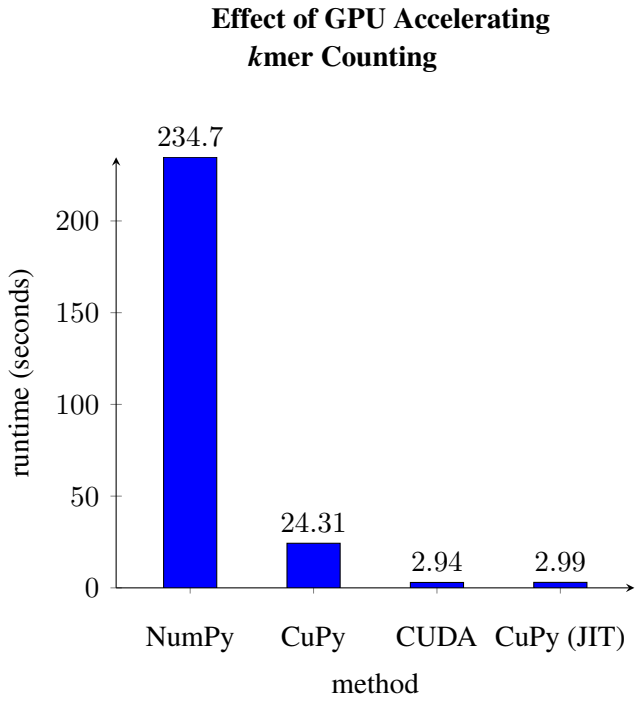



Figure 20: Elapsed time counting the occurrences of 50 million unique *k*mers in a set of 20 million reads, each of 150 bases.

As can be seen in table 3, the CuPy just-in-time compiled method of GPU accelerating *k*mer counting resulted in (close to) identical performance as the CUDA hash table implementation, all in spite of being implemented directly in Python using CuPy's custom kernel support. The memory usage using the CuPy just-in-time compiled solution was also practically identical to that of the CUDA hash table.

## 4.4 GPU Accelerating *k*mer Hashing

The *k*mer hashing component of KAGE is responsible for reading genomic reads from FASTA, FASTQ or other file types, encoding the reads as 2-bit encoded data and finally hashing all valid *k*mers from the 2-bit encoded reads. In KAGE, the final product yielded by this component is a 64-bit unsigned integer array where each element is a 2-bit encoded 31-mer represented in the right-most 62 bits of the integer. Since the number of valid *k*mers in a typical FASTA or FASTQ from sequencing a human genome is extremely vast, the file is usually read, encoded and hashed in chunks to alleviate the required amount of memory. Functionality for this exists in the BioNumPy Python package. BioNumPy implements this using NumPy and some parts of npstructures, resulting in an efficient solution that relies on NumPy's fast array operations for its performance.

### 4.4.1 Replacing NumPy with CuPy

The Python package BioNumPy, which provides an implementation for *k*mer encoding and hashing directly in Python, is built on top of NumPy and npstructures. We utilized the method we described in section 4.2 and used CuPy as a drop-in replacement for NumPy to add further GPU acceleration support to npstructures, and to add partial GPU support to BioNumPy. The resulting pipeline remained the same, but with two notable differences.

1. The raw byte chunk read from the FASTA files would be copied directly to the GPU.
2. The chunk of data received on the GPU would go through the same pipeline of array operations to 2-bit encode and hash the *k*mers, but on the GPU as opposed to the CPU.

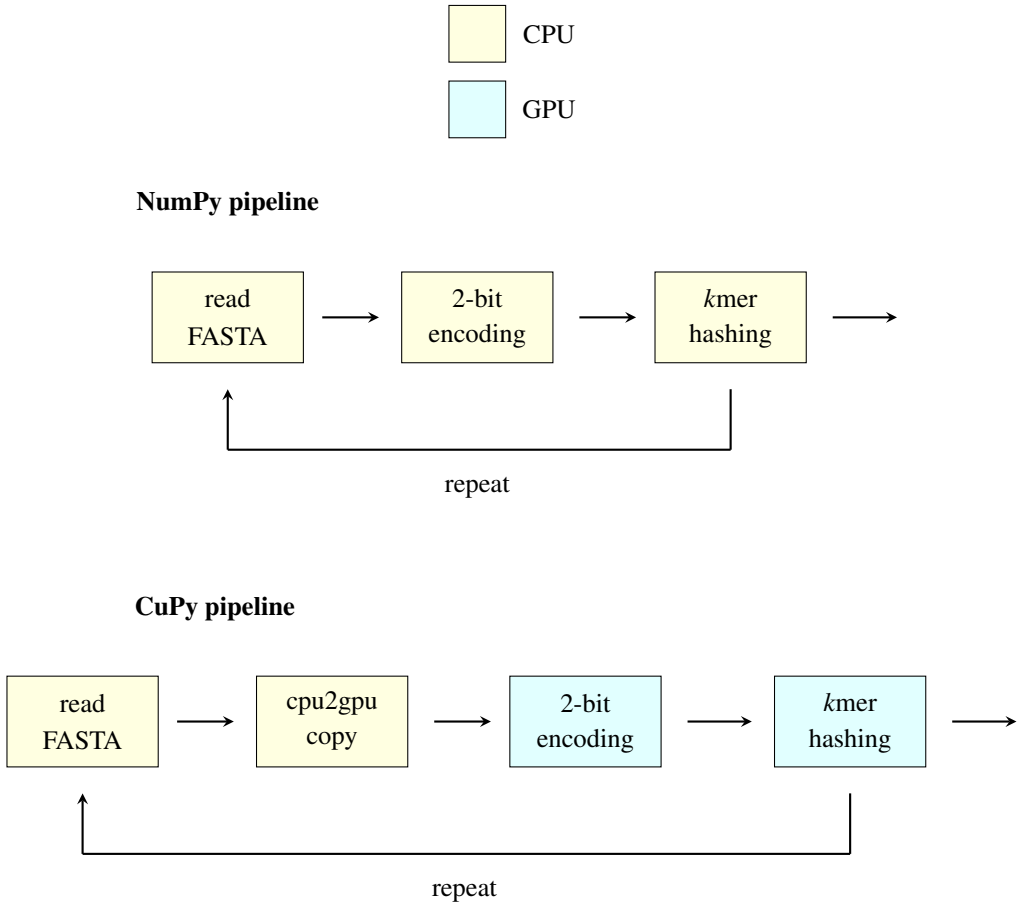



Figure 21: By using CuPy as a drop-in replacement to add GPU acceleration to KAGE's *k*mer hashing step, we effectively introduced a new step in the pipeline: copying the raw text data read from the FASTA file directly to the GPU to allow for all of BioNumPy's array operations that constitute 2-bit encoding and hashing the *k*mers to be performed on the GPU.

Upon completing the 2-bit encoding and *k*mer hashing of the chunk, the resulting *k*mer array already resides in GPU memory for the counting step. Previously, when the 2-bit encoding and *k*mer hashing was performed using only the CPU, the resulting *k*mer array would need

to be copied to the GPU for counting when deploying the GPU accelerated *k*mer counting functionality detailed in section 4.3.

### 4.4.2 Assessment

In order to assess the effects of GPU accelerating *k*mer hashing, we measured the total runtime of reading, 2-bit encoding and hashing every valid *k*mer from a FASTA file containing 20 million reads, each of length 150. The FASTA was read in chunks of 10 million bytes. We benchmarked this runtime using BioNumPy with the standard NumPy backend, and again using CuPy as the backend to compare. The benchmarking yielded the following results:

| **Method** | **Hashing time (seconds)** |
|---|---|
| NumPy | 62.46 |
| CuPy | 5.1 |

Table 4: The runtimes found by hashing all valid *k*mers from a set of 20 million DNA reads, each of length 150. The FASTA file containing the reads was read in chunks of 10 million bytes, and the BioNumPy library in Python was used to read, 2-bit encode and hash each chunk. We ran the benchmark both using NumPy as the backend-library for BioNumPy, meaning the hashing was performed entirely on the CPU using a single thread, and using CuPy as the backend-library for BioNumPy, meaning the raw data read from the FASTA file was copied to the GPU and then 2-bit encoded and hashed there.

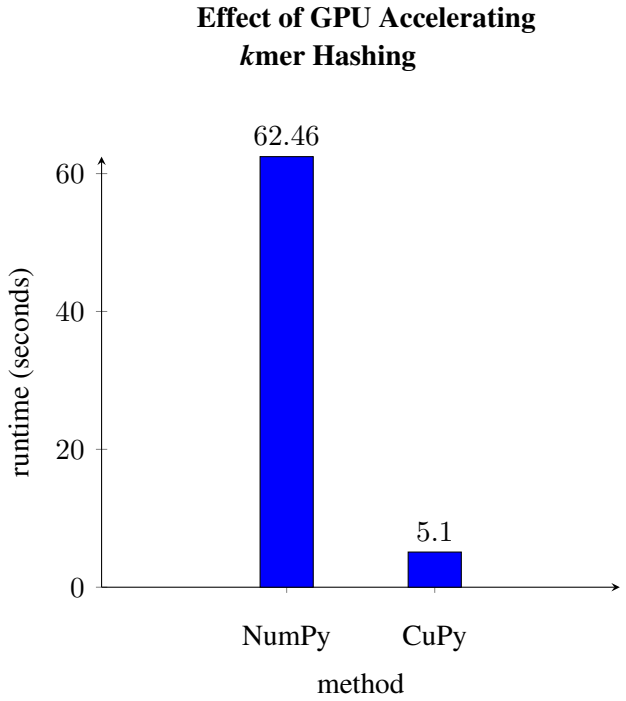



Figure 22: Elapsed time spent reading, 2-bit encoding and hashing all valid *k*mers from 20 million reads in 10 million byte chunks using both NumPy as CuPy to perform the processing on the CPU and GPU respectively.

As can be seen in table 4, GPU accelerating BioNumPy's *k*mer hashing functionality, simply

by using CuPy as a drop-in replacement for NumPy, yielded a more than 12X speedup when hashing every valid *k*mer in a set of 20 million reads of length 150. This dramatic increase in efficiency is achieved despite having to copy each chunk of raw data read from the FASTA file to the GPU memory before processing can begin. Recall that in section 4.2.3, we justified only benchmarking the time spent counting *k*mers on the GPU without including the overhead of copying the *k*mer chunks to the GPU memory. Combined with the *k*mer hashing component's GPU acceleration, which yields a dramatic increase in efficiency despite including this overhead, *k*mer chunks that are hashed on the GPU will already reside in the GPU's memory for counting.

## 4.5 GPU Accelerating Genotyping

The genotyping step of KAGE (introduced in section 3.6) computes genotype-probabilities, supported by the *k*mer counts found during the *k*mer counting portion. A central part of this step is computing the *logarithm of the probability mass function* (LOGPMF) a large number of times on large arrays. Because the LOGPMF computations took up most of the runtime in KAGE's genotyping step and its implementation relied on computing many parallel mathematical expressions for large arrays, we chose to focus on LOGPMF to GPU accelerate the genotyping step.

The logarithm of the probability mass function at $k$ is computed using:

$$logpmf = k * \log(\mu) - \mu - \ln(|\Gamma(k+1)|) \tag{4}$$

where $\Gamma$ is the gamma function. In KAGE, this formula is computed for every element at every index in two equal-sized arrays $k$ and $\mu$. The original implementation utilizes NumPy's efficient array-operations to compute LOGPMF efficiently using data-parallelism on the CPU.

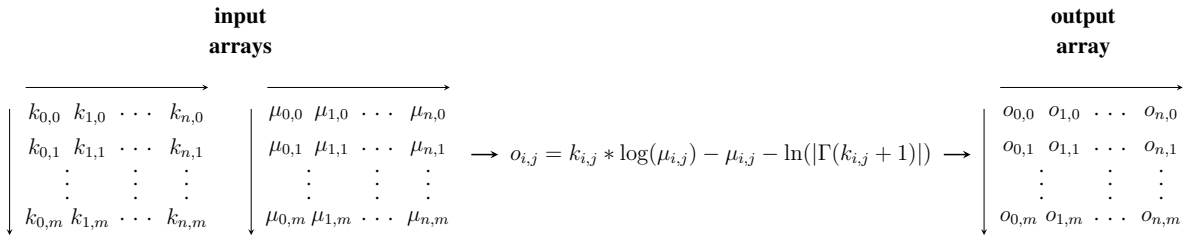



Figure 23: The LOGPMF computation in KAGE is performed on large arrays, relying on NumPy's efficient and data-parallel solutions to run efficiently on the CPU.

### 4.5.1 Replacing NumPy with CuPy

Since the original KAGE solution for computing LOGPMF was implemented using NumPy, our first GPU acceleration was to use the method described in section 4.2.1 where we used CuPy as a drop-in replacement for NumPy to GPU accelerate existing NumPy-based code. The NumPy solution from KAGE relies on SciPy [21] to compute the natural logarithm of the gamma function: $\ln(|\Gamma(x)|)$. For our CuPy solution, we instead used CuPy's cupyx.scipy's implementation.

logpmf.py

```
1  import numpy as np
2  import cupy as cp
3
4  # Original function used in KAGE. Assumes k and r are NumPy arrays
5  def numpy_poisson_logpmf(k, r):
6    return k * np.log(r) - r - scipy.special.gammaln(k+1)
7
8  # Assumes k and r are CuPy arrays
9  def cupy_poisson_logpmf(k, r):
10   return k * cp.log(r) - r - cupyx.scipy.special.gammaln(k+1)
```

Figure 24: The original NumPy-based solution used in KAGE for computing LOGPMF, and a CuPy-based version for GPU acceleration. The CuPy-based GPU accelerated version assumes both input arrays, k and r, are residing in GPU memory.

### 4.5.2 Assessment

In order to assess the effects of GPU accelerating KAGE's LOGPMF function by using CuPy, we benchmarked both functions by computing LOGPMF for input arrays $k$ and $r$. The $k$ array was initialized with 40 million 32-bit integers, randomly set to values between 1 and 100. $r$ was initialized with 40 million 64-bit floating point numbers, randomly set to values between 1 and 100.

The CuPy-based GPU accelerated function presented in figure 24 requires both input arrays, $k$ and $r$, to already be allocated in GPU memory. Benchmarking only the function calls as presented therefore ignores the overhead of copying both $k$ and $r$ to GPU memory before the function call can be made. Thus, for completion, we did not only benchmark both the NumPy and CuPy solutions. We additionally benchmarked the CuPy solution where $k$ and $r$ start off in CPU RAM and are copied to GPU memory before processing, and the result array is copied

back to CPU RAM afterwards. This way, we accounted for the overhead of copying data to and from the GPU.

Our benchmarking yielded the following results:

| **Method** | **Time (milliseconds)** |
|---|---|
| NumPy | 606.94 |
| CuPy (with copy) | 201.9 |
| CuPy (without copy) | 82.49 |

Table 5: Time spent computing the logarithm of the probability mass function for 40 million elements. **NumPy** uses the NumPy-based solution with CPU data-parallelism and a single core. **CuPy (with copy)** uses the CuPy-based solution with GPU acceleration, but $k$ and $r$ are both copied from RAM to GPU memory before processing begins, and finally the result array is copied from GPU memory back to RAM. **CuPy (without copy)** uses the CuPy-based solution with GPU acceleration, but $k$ and $r$ are both already residing in GPU memory, and the result array is not copied back to RAM. In other words, we only measure the processing time of computing the 40 million LOGPMF values. Runtimes are the mean of 100 runs.

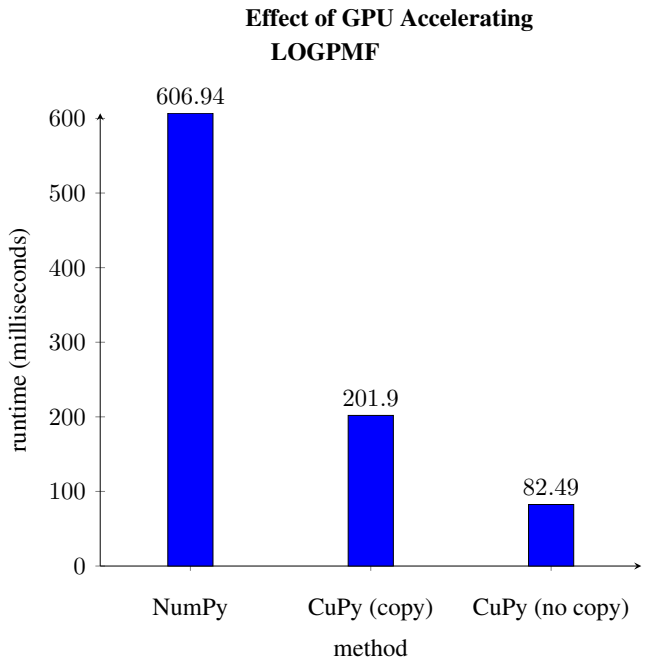



Figure 25: Time (milliseconds) spent computing the LOGPMF for input arrays of 40 million elements.

### 4.5.3 Implementing LOGPMF using CUDA

While the method of using CuPy as a drop-in replacement for NumPy provided a significant speedup when computing the LOGPMF, it reveals a problem that arises when using NumPy and CuPy in Python. When computing a nested expression, such as when computing the LOGPMF shown in equation 4, the expression will be computed and evaluated following Python's evaluation rules, subsequently creating several temporary arrays. This means that array allocations,

de-allocations, memory writes and reads are happening needlessly while evaluating the Python expression.

By moving into C++ and using CUDA, we can create our own CUDA kernel to compute the LOGPMF for large arrays, avoiding these redundant operations. We implemented a simple kernel where we avoid needlessly storing sub-expressions in temporary arrays.

logpmf_kernel.cu

```
__global__ void logpmf_kernel(
    const int *k, const double *r, double *out, const int size) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < size) {
    out[i] = k[i] * logf(r[i]) - r[i] - lgammaf(k[i] + 1);
  }
}
```

Figure 26: The CUDA kernel implemented for computing the LOGPMF for arrays on the GPU.

### 4.5.4 Assessment

To benchmark the CUDA kernel implementation we used the same benchmark as for the CuPy solution described in section 4.5.2. Our benchmarking yielded the following results:

| **Method** | **Time (milliseconds)** |
|---|---|
| NumPy | 606.94 |
| CuPy (with copy) | 201.9 |
| CuPy (without copy) | 82.49 |
| CUDA (with copy) | 68.33 |
| CUDA (without copy) | 4.88 |

Table 6: Time spent computing the logarithm of the probability mass function for 40 million elements. **NumPy**, **CuPy (with copy)** and **CuPy (without copy)** are the benchmark results from table 5. **CUDA (with copy)** uses the CUDA kernel implementation, but $k$ and $r$ are both copied from RAM to GPU memory before the kernel is launched, and the result array is copied from GPU memory back to RAM. **CUDA (without copy)** uses the CUDA kernel implementation, but $k$ and $r$ are both already residing in GPU memory and the result array is not copied from GPU memory to RAM. Runtimes are the mean of 100 runs.

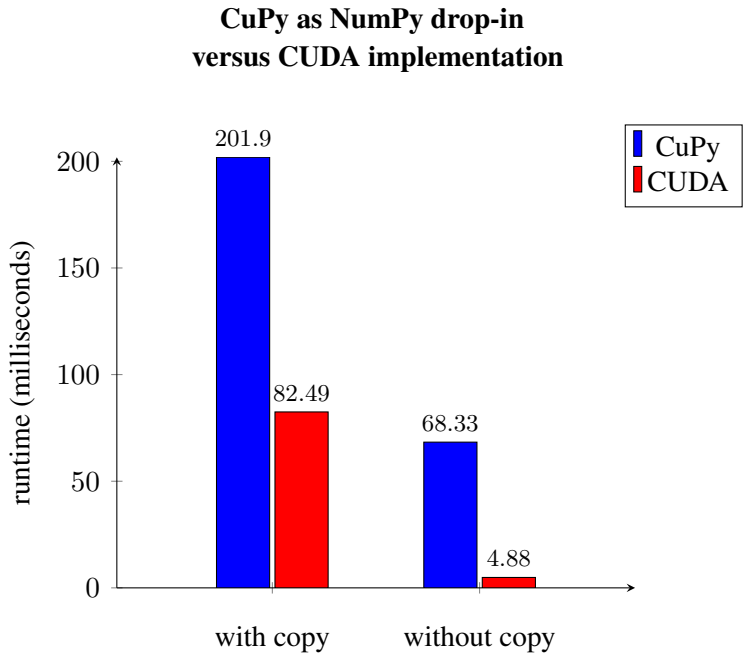



Figure 27: Time (milliseconds) spent computing the LOGPMF for input arrays of 40 million elements.

As can be seen in table 6, implementing our own CUDA solution, alleviating the wasteful array allocations and memory reads and writes, resulted in a significant speedup.

### 4.5.5 Extending the LOGPMF Implementation

Despite already having achieved significant speedup using both CuPy and CUDA to GPU accelerate LOGPMF, further examination of KAGE revealed that more of the context around the LOGPMF function call was eligible to be GPU accelerated. The original part of KAGE that made calls to the LOGPMF function is presented below:

kage/sampling_combo_model.py

```
1 def extended_logpmf(observed_counts, counts):
2     sums = np.sum(counts, axis=-1)[:, None]
3     frequencies = np.log(counts / sums)
4     poisson_lambda = (np.arange(counts.shape[1])[None, :] +
          ERROR_RATE) * BASE_LAMBDA
5     prob = logpmf(observed_counts[:, None], poisson_lambda) # LOGPMF
          call is made here
6     prob = logsumexp(frequencies + prob, axis=-1)
7     return prob
```

Figure 28: The function inside the KAGE genotyper implementation that uses the LOGPMF function. The outer function also relies on NumPy array operations for performance.

We used two previously described methods to implement the outer function from figure 28 with GPU acceleration: we used CuPy as a drop-in replacement for NumPy, and we implemented a CUDA kernel in C++ where we again were able to optimize away unnecessary array allocations and de-allocations, as well as RAM writes and reads. The implementations can be found (along with the GPU accelerated LOGPMF function) online at https://github.com/jorgenwh/custats.

### 4.5.6 Assessment

To assess the extended LOGPMF implementation, we benchmarked the NumPy-based version found in KAGE, the CuPy drop-in replacement for NumPy version, and the CUDA kernel implementation. For both GPU accelerated methods, namely the CuPy and CUDA methods, we additionally benchmarked their performances when inputs are copied from RAM to GPU memory beforehand and results are copied from GPU memory to RAM.

To benchmark, we initialized the input arrays *observed_counts* and *counts* (the model counts) to be randomly generated numbers. *observed_counts*, a one-dimensional array, contained 5 million 32-bit integers. *counts*, a two-dimensional array, contained 5 million rows of length 15, totalling 75 million 16-bit floating point numbers, except when running the CUDA version since we did not implement support for half-precision.

Our benchmarking yielded the following results:

| **Method** | **Time (milliseconds)** |
|---|---|
| NumPy | 2242.9 |
| CuPy (with copy) | 328.44 |
| CuPy (without copy) | 293.24 |
| CUDA (with copy) | 56.59 |
| CUDA (without copy) | 5.92 |

Table 7: Benchmarking results for the extended LOGPMF function. The results were determined from averaging the runtime of the NumPy implementation in KAGE, the same implementation with CuPy as backend array-library, both with and without including the time spent copying input arrays to the GPU and the result array from the GPU back to RAM, and the CUDA solution we implemented, both with and without copy times. The NumPy solution running on the CPU does not need to account for copying data to and from the GPU. Thus, no value is present for the 'with copy' bar for NumPy.

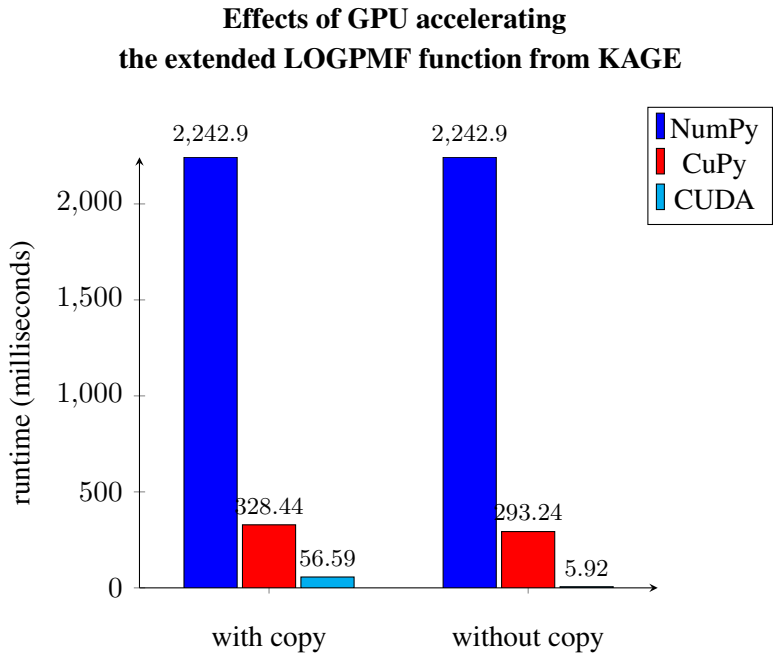



Figure 29: Benchmarking results for the extended LOGPMF function. The NumPy solution running on the CPU does not need to account for copying data to and from the GPU, thus, their runtimes are identical.

As can be seen in table 7, both the methods we tried for GPU accelerating the extended LOGPMF function found in KAGE yielded significant runtime speedup. Clearly, our CUDA implementation performs significantly better than simply using CuPy as a drop-in replacement for NumPy.

# 5 Results

In the following section we will present the final GPU accelerated version of KAGE: GKAGE. Additionally, we will benchmark GKAGE against KAGE on two different computer systems to evaluate the final speedup achieved by GPU accelerating KAGE.

## 5.1 GKAGE

In the previous section we explored three different methods for GPU accelerating existing NumPy-based code in Python: 1) using CuPy as a drop-in replacement for NumPy, 2) implementing our own GPU accelerated solutions in C++ with CUDA and using pybind11 to create Python bindings for said solutions, and 3) using CuPy's custom jit (just-in-time) compiled kernel support to implement kernels directly in Python, achieving similar control and granularity as with method 2.

We chose to integrate the two following GPU accelerated functionalities into KAGE: 1) our custom GPU accelerated hash table implemented in C++ using CUDA with pybind11 Python bindings (described in section 4.3), and 2) the CuPy drop-in solution for *k*mer encoding and hashing (described in section 4.4). The GPU accelerated functionality implemented for the genotyping step in KAGE was not included, as it did not result in faster runtimes. We integrated the solutions into KAGE in such a way that the same piece of software could be run on both the CPU and the GPU. In fact, running GKAGE is achieved by running KAGE, using the -g flag to enable GPU acceleration. KAGE (and GKAGE) can be found at https://github.com/kage-genotyper.

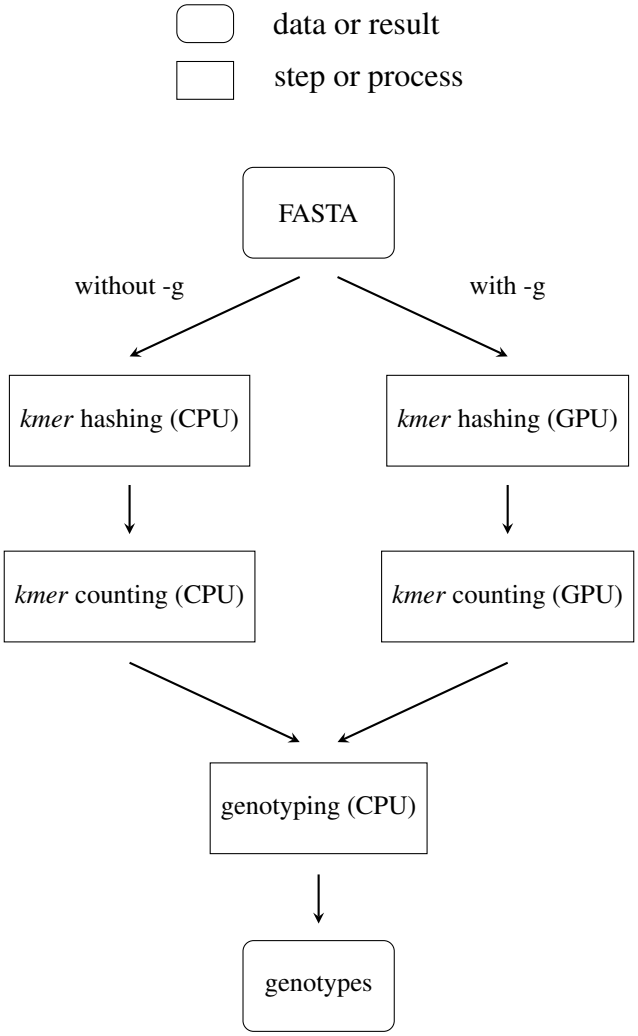



Figure 30: To run GKAGE, you simply run KAGE with the -g flag to indicate that you want to use the GPU to accelerate the processing. Out of the three components that we explored to GPU accelerate, only two made it into GKAGE. Thus, only *k*mer (encoding and) hashing and *k*mer counting is GPU accelerated when running KGAKE. The genotyping step, which constitutes a small portion of the runtime even on the CPU, is identical and runs on the CPU in both KAGE and GKAGE.

## 5.2 Benchmarking

In order to benchmark the effectiveness of the added GPU acceleration in GKAGE, we decided to benchmark GKAGE against KAGE to account for the speedup. The choice of only benchmarking GKAGE against KAGE was made based on the fact that KAGE recently showed that it was an order of magnitude faster than any other known genotyper [4].

### 5.2.1 Snakemake pipeline

In order to adequately benchmark GKAGE, we set up a Snakemake pipeline that runs both KAGE and GKAGE on a full human genome and records the runtimes while also checking that the results of both processes are identical. This Snakemake pipeline can be found at https:

//github.com/kage-genotyper/GKAGE-benchmarking.

### 5.2.2 Systems

We benchmarked GKAGE against KAGE on two systems. **System 1**: a high-end compute server with a 64-core AMD EPYC 7742 CPU and two Nvidia Tesla V100 GPUs. **System 2**: a consumer grade desktop with a 6-core Intel Core i5-11400F CPU and a Nvidia GTX 1660 SUPER GPU.

| **System** | **CPU** | **GPU** |
|---|---|---|
| 1: High-end server | AMD EPYC 7742 | 2x Nvidia Tesla V100 |
| 2: Consumer desktop | Intel Core i5-11400F | Nvidia GTX 1660 SUPER |

Table 8: The two systems used to benchmark GKAGE against KAGE to account for the speedup achieved through GPU acceleration. **System 1** is a high-end compute server with top-of-the-line hardware. **System 2** is a consumer grade desktop gaming computer.

Benchmarking on these two systems allowed us to benchmark GKAGE on both a very performant and a less performant system to examine the effects of GPU acceleration in both instances. A caveat with the high-end server system (system 1) was that it was a publicly available server for students at the University of Oslo. Thus, students were frequently running jobs on the server, occupying both CPU and GPU processing cores and memory. To the best of our capacity, we conducted our benchmarking during periods of lower activity.

### 5.2.3 Runtimes

We benchmarked GKAGE against KAGE on both system 1 and system 2. When running on system 1, we allowed KAGE 16 cores and GKAGE one Nvidia Tesla V100 GPU. When running on system 2, we allowed KAGE 6 cores and GKAGE the Nvidia GTX 1660 SUPER GPU. We ran the Snakemake pipeline, benchmarking GKAGE against KAGE, on both systems and achieved the following results:

| System | KAGE | GKAGE |
|---|---|---|
| 1 | 1993 sec | 178 sec |
| 2 | 510 sec | 94 sec |

Table 9: The resulting runtimes (in seconds) found by benchmarking GKAGE against KAGE on a high-end server and a consumer desktop computer

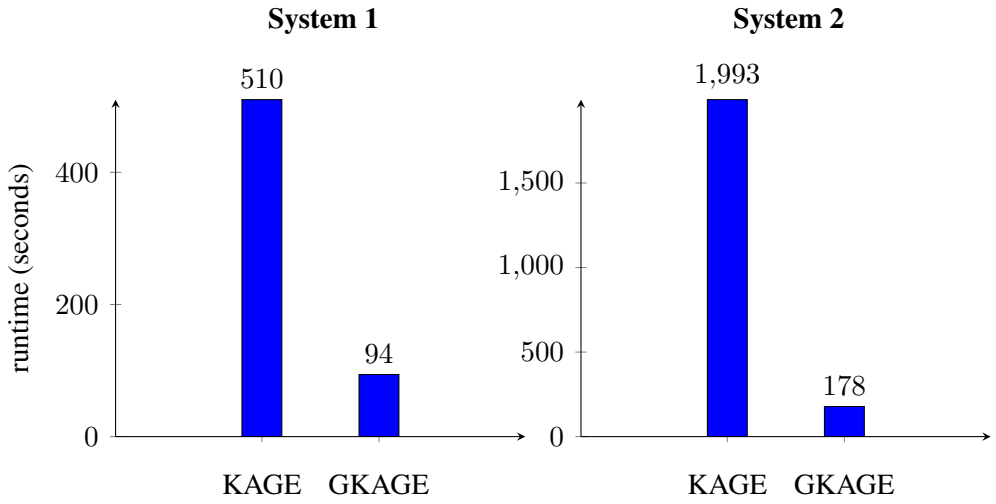



Figure 31: Benchmarking GKAGE against KAGE on both system 1 (high-end server) and system 2 (consumer desktop) revealed that GKAGE could genotype a full human genome more than 5 times faster than KAGE on a high-end server, and more than 11 times faster on a consumer desktop computer.

The results from the benchmarking revealed that GKAGE achieves significant speedup over KAGE when genotyping a full human genome. As seen in table 9 and figure 31, we can see that GKAGE achieves more than 5X speedup over KAGE on a high-end compute server and more than 11X speedup on a consumer grade desktop computer.

## 5.3 GPU Acceleration Methods

In our expedition to GPU accelerate components of the KAGE genotyping pipeline we found three distinct methods for GPU accelerating existing Python code based on NumPy. Each method has its own unique set of advantages and drawbacks that we gained insight into during our project. A more detailed discussion about their advantages and drawbacks will take place in section 6.1.

## 5.4 bioRxiv Preprint

As a part of this master's project, we published a preprint [26] in biorxiv - the preprint server for biology. The preprint, *Ultra-fast genotyping of SNPs and short indels using GPU acceleration*, presents GKAGE as a software tool and briefly details the runtime speedup gained over KAGE and how GKAGE was implemented. The preprint article can be found in appendix A.

# 6 Discussions

## 6.1 Advantages and Drawbacks of Methods

In section 4 we explored three different methods for GPU accelerating existing Python code. Each of the methods we explored have a unique set of advantages and drawbacks. The discipline of designing and implementing GPU programs is quite distinct from the more mainstream discipline of implementing programs meant to run on the CPU. Becoming an expert GPU programmer can in many instances take years with all the technologies and tools available today. Thus, an interesting dimension when assessing the advantages and drawbacks of the GPU acceleration methods we have explored is how seamless it is to implement the solution, particularly for someone with little or no experience in GPU programming. Other metrics we will discuss include how seamless the integration of a solution into Python is, how quickly a solution can be implemented compared to the alternative methods, among other per-method details.

### 6.1.1 Using CuPy as a NumPy Drop-in Replacement

The first GPU acceleration method we explored in section 4.2, was to use CuPy, a GPU accelerated array library with an interface designed to closely follow NumPy, as a drop-in replacement for NumPy to GPU accelerate an already existing solution that was based on NumPy for performance.

#### 6.1.1.1 Advantages

This method is ideal in cases where an already existing Python solution based on NumPy exists and it is interesting to see whether porting it to the GPU will likely be beneficial. The technique of using CuPy as a drop-in for NumPy is, by a large margin, the fastest way of implementing GPU acceleration. This does, however, require an already existing solution based on NumPy. If a solution is to be made from scratch, CuPy is still an adequate tool for "quick and dirty" testing with seamless GPU acceleration directly in Python. Additionally, this method does not require deep knowledge or understanding of the GPU architecture, although understanding the GPU's hardware is helpful when determining which NumPy solutions are good candidates for GPU acceleration through CuPy.

#### 6.1.1.2 Drawbacks

While an advantage of CuPy is that it allows for seamless "quick and dirty" testing directly in Python, this conversely also introduces a drawback of this method. Many solutions imple-

mented using NumPy are designed for fast processing on the CPU. While they can often be great candidates for the GPU since they perform array operations that can be well suited for the GPU architecture, this match is not guaranteed. In such cases, the GPU will provide inadequate performance boosts, even if the solution could be redesigned to better fit the GPU architecture.

### 6.1.2 Custom C++ Implementations Using CUDA

The second GPU acceleration method we explored in section 4.3, was to implement our own solution directly in C++ using the CUDA framework. We then used pybind11 to create Python bindings for our C++ functionality to gain access directly inside Python.

#### 6.1.2.1 Advantages

The clearest advantage of this method is the control over hardware and granularity achieved when implementing the solution directly using Nvidia's programming platform: CUDA. This provides the possibility of more closely tailoring the implementation to the problem and using all of CUDA's technology to gain the best performance possible. Additionally, since such an implementation would be created using C++, this allows access to C++ features that are otherwise out of reach in Python, such as thread parallelization.

#### 6.1.2.2 Drawbacks

For this method too, its main advantage also yields its greatest drawback. The CUDA programming platform is vast and provides support that can be extremely useful to solve certain problems effectively, but that also requires deep knowledge and understanding of the GPU hardware- and programming model. Additionally, since CUDA features are used in C++, the programmer would need to be, at the very least, comfortable with writing software in C++. Since such a solution would be implemented in C++, this makes integration into Python more difficult. Bindings using ctypes, pybind11 or some other method would be necessary to gain access to the solution directly in Python. C++ is also a significantly more verbose language, largely due to its more fine-grained control. The implementation is in many ways forced to be more detailed, which results in more time required to create the implementation.

### 6.1.3 Custom JIT-Compiled Kernels in Python

The third and final GPU acceleration method we explored in section 4.3.3, was to implement our solution using CuPy's support for JIT compiled (just-in-time-compiled) custom kernels. CuPy, directly in Python through its module interface, provides access to certain CUDA functionality as well as support for creating kernels directly in Python code that are then compiled

when the running program first encounters them.

#### 6.1.3.1 Advantages

This method has many of the same advantages as the method discussed in 6.1.1, and can be viewed as an extension of the CuPy drop-in for NumPy method. What it brings in addition to being a drop-in replacement for NumPy is its JIT compiled custom kernel support and, although limited, CUDA functionality support. This allows for simple usage, similar to NumPy, where it is suitable, and more detailed usage such as implementing custom kernels in cases where the straight-forward array-interface does not suffice. In section 4.3.3, we showed that we could, in full, re-implement our CUDA based hash table directly using CuPy's custom kernel support. This was all achieved while never leaving Python, meaning a programmer does not need to know C++ in order to implement custom kernels this way. More additional functionality not used in our implementation of GKAGE is also supported through CuPy, such as CUDA streams cooperative groups and more [27].

#### 6.1.3.2 Drawbacks

While it is helpful to be able to implement custom kernels directly in Python code, it is uncertain how much simplicity this in effect introduces. The kernels implemented using CuPy's custom kernel support still need to adhere to the same programming model as CUDA kernels implemented in C++. A programmer with little to no knowledge of how the GPU hardware and programming model works, will not with ease be able to implement effective kernels this way. Therefore, we would argue that this advantage does not do much more than alleviate the need to delve into C++ and set up proper compiling and Python bindings.

## 6.2 Drawbacks of Graphical Processing Units

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

The reason why GPUs are so powerful when it comes to accelerating certain parallel programs, is because they are designed for such problems. GPUs are highly specialized compute accelerators that perform poorly when applied to any problem that does not fit its compute architecture. Additionally, today's GPUs are expensive and less accessible than traditional CPUs. Because of the CPU's flexibility in regards to the problems it can be used to solve, CPUs exist in most - if not all - existing computers today. GPUs however, are less common, particularly serious GPUs fitting for scientific computing.

Furthermore, GPU programming is by many considered to be its own discipline, as the programming models and paradigms used when developing GPU programs are quite different from the typical sequential programs written for the CPU. This leads to a higher bar for entry when it comes to developing effective GPU programs for difficult problems, as fewer expert programmers have the knowledge and training to understand such systems.

## 6.3 Further Work

While GKAGE demonstrates that alignment-free genotyping can be sped up through GPU acceleration, we believe that the current GKAGE runtimes can still be significantly improved. We will here identify some possible avenues where the current implementation can be altered or improved in order to better use the hardware and technology available to gain more speedup.

### 6.3.1 Better GPU Hash Table

While the GPU hash table used in GKAGE for *k*mer counting is plenty effective for our purpose of demonstrating the effectiveness of GPU acceleration in alignment-free genotyping, alternative hash tables exist that, with correct integration, should perform better. Since optimizing GPU hash tables is all but a discipline in and of itself and beyond the scope of this thesis, our GPU hash table implements a naive solution for collision handling and probing. An interesting future avenue would be to integrate a version of a bucketed static cuckoo hash table from BGHT [28] to evaluate how a state-of-the-art hash table would perform compared to our own. Although faster, we doubt the effect would be dramatic in our case.

### 6.3.2 Parallelization of *k*mer Chunk Preparation

Recall that in the *k*mer counting step in KAGE, the input FASTA file is read in chunks. Each chunk of data is then 2-bit encoded and all valid *k*mers are hashed from the 2-bit encoded data chunk. Finally, the chunk of hashed *k*mers are counted. In GKAGE, the 2-bit encoding, *k*mer hashing and *k*mer counting are performed on the GPU. Thus, the chunk of data read from the FASTA file is copied to the GPU's memory before processing begins. Currently, GKAGE performs all of these processes sequentially.

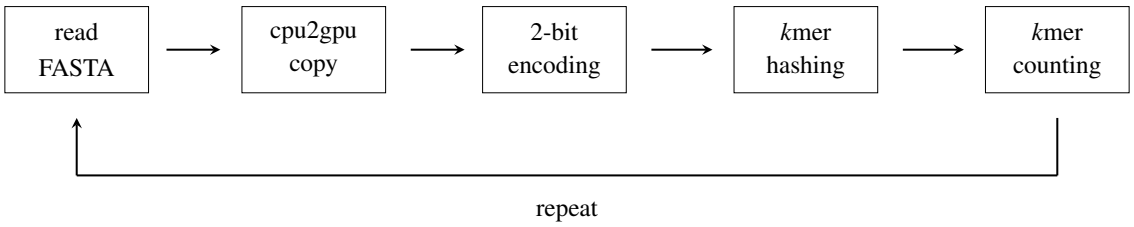



Figure 32: GKAGE's current *k*mer counting pipeline performs several steps in a sequential fashion. While these steps are performed on many *k*mers in parallel at once, each chunk (containing many *k*mers) is processed sequentially.

By utilizing CUDA streams we can parallelize the copying of data to the GPU and the processing of the previous chunk, creating a new and more efficient pipeline.

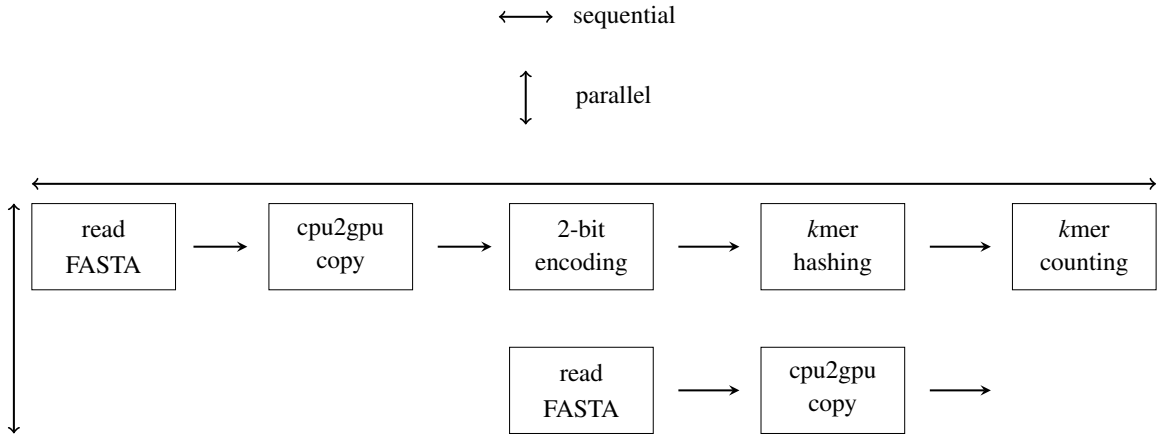



Figure 33: An illustration of a more optimal *k*mer counting pipeline. By utilizing CUDA's streams, we can parallelize copying of data to the GPU and actual GPU processing.

# 7 Conclusion

# References


[1] National Human Genome Research Institute. *The Cost of Sequencing a Human Genome*. https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost. Accessed: 2023-04-24. 2023.

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

[3] 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.

[4] 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>.

[5] Adam Auton et al. "A global reference for human genetic variation". In: *Nature* 526.7571 (Oct. 2015), pp. 68–74.

[6] 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>.

[7] 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>.

[8] 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.

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

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

[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] 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>.

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

[14] 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.

[15] 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.

[16] 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>.

[17] Thomas Alsop. *PC graphics processing unit (GPU) shipment share worldwide from 2nd quarter 2009 to 4th quarter 2022, by vendor*. https://www.statista.com/statistics/754557/worldwide-gpu-shipments-market-share-by-vendor/. Accessed: 2023-05-7. 2023.

[18] NVIDIA. *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 2023-05-7. 2023.

[19] Dave Kuhlman. *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. https://www.davekuhlman.org/python_book_01.pdf. Accessed: 2023-04-28. 2023.

[20] Stack Overflow. *Most popular technologies*. https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language-prof. Accessed: 2023-04-28. 2023.

[21] 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.

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

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

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

[25] Austin Appleby. *MurmurHash3*. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp. Accessed: 2023-04-20. 2023.

[26] Jørgen Wictor Henriksen et al. “Ultra-fast genotyping of SNPs and short indels using GPU acceleration”. In: *bioRxiv* (2023). DOI: 10.1101/2023.04.14.534526. eprint: https://www.biorxiv.org/content/early/2023/04/14/2023.04.14.534526.full.pdf. Address <https://www.biorxiv.org/content/early/2023/04/14/2023.04.14.534526>.

[27] Ryosuke Okuta et al. “CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations”. In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. 2017. Address <http://learningsys.org/nips17/assets/papers/paper_16.pdf>.

[28] Muhammad A. Awad et al. “Analyzing and Implementing GPU Hash Tables”. In: *SIAM Symposium on Algorithmic Principles of Computer Systems*. APOCS23. Jan. 2023. Address <https://escholarship.org/uc/item/6cb1q6rz>.

# Appendices

## A bioRxiv Preprint - *Ultra-fast genotyping of SNPs and short indels using GPU acceleration*

# Ultra-fast genotyping of SNPs and short indels using GPU acceleration

## Authors


- **Jørgen Wictor Henriksen**
  Biomedical Informatics research group, Department of Informatics, University of Oslo, Oslo, Norway

- **Knut Dagestad Rand**
  Biomedical Informatics research group, Department of Informatics, University of Oslo, Oslo, Norway; Centre for Bioinformatics, University of Oslo, Oslo, Norway

- **Geir Kjetil Sandve**
  Biomedical Informatics research group, Department of Informatics, University of Oslo, Oslo, Norway; Centre for Bioinformatics, University of Oslo, Oslo, Norway; UiORealArt Convergence Environment, University of Oslo, Oslo, Norway

- **Ivar Grytten** ✉
  Biomedical Informatics research group, Department of Informatics, University of Oslo, Oslo, Norway

✉ — Correspondence to Ivar Grytten <ivargry@ifi.uio.no>.


Ultra-fast genotyping of SNPs and short indels using GPU acceleration

## Abstract


As decreasing DNA sequencing costs leads to a steadily increasing rate of generated data, the development of efficient algorithms for processing of the sequence data is increasingly important to reduce costs and energy consumption. Recent work have shown that genotyping can be done efficiently and accurately using alignment-free methods that are based on analyzing kmers from sequenced reads. In particular, we have recently presented the KAGE genotyper, which uses an efficient pangenome representation of known individuals in a population to further increase accuracy and efficiency. While existing genotypers like KAGE use the Central Processing Unit (CPU) to count and analyze kmers, the Graphical Processing Unit (GPU) has shown promising for reducing runtime for similar problems.

We here present GKAGE, a new and improved version of KAGE that utilizes the GPU to further increase the computational efficiency. This is done by counting and analyzing large amounts of kmers in the many parallel cores of a GPU. We show that GKAGE is, on hardware of comparable cost, able to genotype an individual up to an order of magnitude faster than KAGE while producing the same output, which makes it by far the fastest genotyper available today. GKAGE can run on consumer-grade GPUs, and enables genotyping of a human sample in only a matter of minutes without the need for expensive high-performance computers. GKAGE is open source and available at https://github.com/kage-genotyper/kage.

# Introduction

The cost of sequencing a full human genome has fallen drastically in recent years, and consumers can now get their whole genome sequenced for a few hundred dollars [1], a fraction of what was the price only a few years ago. As millions of genomes are likely to be sequenced in the coming years, there is an ever-increasing need for efficient methods for analyzing this genomic data. At the core of such analysis is variant detection, determining which genetic variants are present in a sample based on the sequenced reads.

Recent methods [2,3,4,5] have shown that detecting variants in a human sample can be performed efficiently and with high accuracy by genotyping the sample against an existing database of known human variation. Such methods use prior knowledge from e.g. the 1000 Genomes Project [6] about where in the genome individuals frequently have variation, and for each such genetic variant use the genomic reads from the sample to infer the most likely genotype. While such methods have traditionally been based on aligning reads to a reference genome, which is slow, recent alignment-free methods have shown that drastic speedup can be achieved by instead analyzing kmers from the sequenced reads. In a recent publication [2], we proposed a highly efficient alignment-free method, KAGE, that uses prior knowledge from a population to achieve high genotyping accuracy while being more computationally efficient than other alignment-free genotypers. Alignment-free genotypers, like KAGE, generally rely on analyzing kmers from reads against kmers associated with genomic variants. Prior to genotyping, kmers that represent alleles of variants of interest are collected and stored in an index (e.g. a hashmap) that enables lookup of kmers from reads to variant alleles. This is typically done once for a set of variants, and this index can then be reused for genotyping any individual against those variants. When genotyping an individual, kmers from reads are collected and looked up in the kmer index, to obtain kmer counts for each variant allele. Generally, genotype probabilities are then found by analyzing the counts of how many reads support the various alleles of the variants of interest. In KAGE, these counts are combined with expected counts sampled from a population to obtain better estimates.

In contrast to other genotypers, especially alignment-based genotypers like GATK [7], the time-consuming steps of KAGE (like processing and counting kmers) can readily benefit from the massive parallelization that the Graphical Processing Unit (GPU) offers [9]. Here, we show that applying GPU acceleration to the compute-heavy parts of KAGE leads to a genotyper GKAGE of unsurpassed computational efficiency. GKAGE uses the GPU to process genomic reads, extract kmers and count the number of kmers that support alleles of genetic variants. We show that this results in a substantial speedup of up to 10X over the original KAGE genotyper (which was already faster than any other genotyper), while producing the exact same output. GKAGE has been implemented so that it is able to run even on standard consumer GPUs, and is able to genotype a whole human sample in a matter of minutes.

# Results

GKAGE is a GPU-accelerated version of the recently published KAGE genotyper. GKAGE uses CUDA-enabled GPUs to efficiently parse and encode kmers from reads, to genotype a set of known SNPs and indels based on the kmer counts. GKAGE produces output that is identical to that of KAGE, with reduced runtime on systems that support the CUDA interface for GPU acceleration. The software is open source and available at https://github.com/kage-genotyper/kage. As part of GKAGE, we have also implemented a static GPU hashmap for counting kmers through a Python interface, available at https://github.com/kage-genotyper/cucounter.

We have recently shown that KAGE is an order of magnitude faster than existing genotypers while giving better or comparable accuracy [2]. We thus only benchmark GKAGE against KAGE to show the effect of GPU acceleration. We do this by running GKAGE and KAGE on a human whole genome sample (15x coverage) on two different systems:

1. A high-performance server with an AMD EPYC 7742 64-Core CPU and two NVIDIA Tesla V100 GPUs. KAGE was run using 16 threads and GKAGE was run using one GPU.

2. A regular desktop computer with an 11th Gen Intel(R) Core(TM) i5-11400F @ 2.60GHz CPU with 6 cores and a NVIDIA GTX 1660 super GPU. KAGE was run using 6 threads.

Table 1 shows the runtimes on these two systems. GKAGE is approximately 5x faster on the high-end system and more than 10x faster on the desktop computer. Since GKAGE only needs the kmer counts of a predefined set of kmers (those associated with variants), and no existing GPU-based kmer counter is able to count only a given set of kmers, we have implemented our own kmer counter as part of GKAGE. An alternative solution would be to count all kmers using an existing tool and filter out those kmers that are relevant. Table 1 also shows the time spent by the GPU kmer counter Gerbil [8] to only count kmers.

**Table 1:** Running times of KAGE, GKAGE and Gerbil (only kmer counting)

| | **KAGE** | **GKAGE** | **Gerbil (only kmer counting)** |
|---|---|---|---|
| Desktop computer | 1993 sec | 178 sec | 464 sec |
| High-performance computer | 510 sec | 94 sec | 438 sec |

# Methods

## Implementation

Here we describe more in detail how GKAGE has been implemented. While GKAGE is implemented as part of KAGE, and shares large parts of its code with KAGE, compute-heavy parts have been reimplemented so that the GPU is utilized. GKAGE implements GPU support for two bottleneck components of KAGE that were suitable for GPU acceleration:

**Reading and encoding kmers** from a FASTA/FASTQ file is achieved in KAGE by using BioNumPy [10], a Python library built on top of NumPy [11]. BioNumPy uses NumPy to efficiently read chunks of DNA reads from fasta files, encode the bases to a 2-bit representation, and then encode the valid kmers as 64-bit integer representations in an array. GPU support for this step was achieved by utilizing CuPy [12], a GPU accelerated computing library with an interface that closely follows that of NumPy. This component was implemented in GKAGE by replacing the NumPy module in BioNumPy with CuPy, effectively replacing all NumPy function calls with calls to CuPy's functions providing the same functionality with GPU acceleration. This strategy worked out of the box for most parts of the BioNumPy solution, with only a few custom modifications having to be made due to certain functions in NumPy's interface not being supported by CuPy.

**Counting kmers** As part of GKAGE, we have implemented a static hashtable for counting a predefined set of kmers on the GPU. The implementation supports parallel and high-throughput hashing and counting of large chunks of kmers simultaneously on the GPU. This static hashtable is implemented as a C++ class in CUDA [13], with two arrays of 64-bit and 32-bit unsigned integers to represent kmers (keys) and counts (values), respectively. CUDA kernels are implemented that handle

insertion (only once during initialization of the hashtable), counting and querying of kmers. The hashtable uses open addressing and a simple linear probing scheme with a murmur hash [14] for the keys.

To find the position of a kmer $k$ in the hashtable, the initial probe position $p_0$ is found by computing

$$p_0 = hash(k) \bmod c$$

where $hash$ is a murmur hash function and $c$ is the capacity of the hashtable. If $p_0$ is occupied by a different kmer than $k$, the next probing position $p_i$ can be computed given the previous probing position $p_{i-1}$ with

$$p_i = (p_{i-1} + 1) \bmod c$$

The probing will continue until either $k$ or an empty slot in the hashtable is observed (See Figure 1 for an illustration of this).

The hashtable supports three main operations: insertion, counting and querying. In each of the cases, the input is an array of 2-bit encoded kmers. When querying, the return value is an array of counts associated with the input kmers. For insertion, counting or querying of $n$ kmers, $n$ CUDA threads are launched. Each thread is then responsible for fulfilling the relevant operation associated with the kmer, i.e. incrementing or fetching the count associated with the kmer in the hashtable, all achieved using the probing scheme previously described. Furthermore, for insertions and count updates, CUDA atomic operations are used to avoid race conditions. To use the hashtable class in Python, C++ bindings are implemented using pybind11 [15].

Since KAGE only needs to count kmers that are preselected to represent alleles of known variants, which typically is only a small subset of the kmers present in genomic reads, the hashmap needed for this requires only a few gigabytes of memory. Thus, when genotyping 28 million variants of a human sample, a GPU with 4GB of memory is sufficient (see details below).

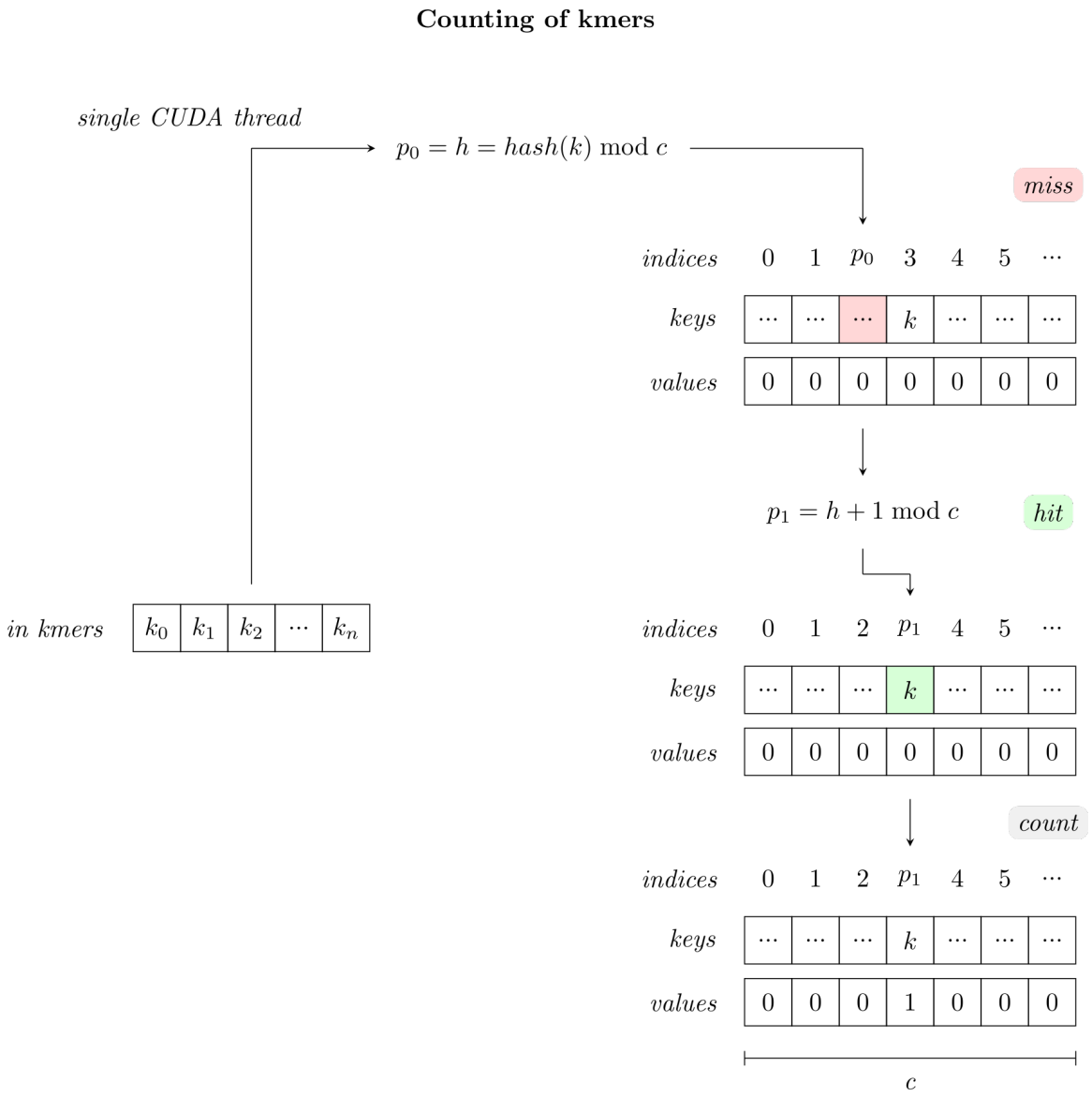



**Figure 1:** As an array of 64-bit integer encoded kmers are counted by the hash table, each CUDA thread will compute the first probe position $p_0$ for each individual kmer, and then continue probing by linearly moving up to the next consecutive slot until either an empty slot or the original kmer handled by the thread is observed. If an empty slot is observed, the thread terminates. If the original kmer is observed, the value at the current slot is increased.

## Memory Usage

The memory usage for GKAGE is influenced by two factors: The size of the kmer index, and the number of sequences to handle at a time (chunk size). Since the chunk size can be set to a suitable number for a given system (we used 10 MB in our experiments), the main factor will be the size of the kmer-index.

The size of the kmer-index itself is again influenced by three factors: The number of variants to be genotyped ($N_v$), the average number of kmers per variant ($k$), and the loading factor of the hash table ($L$). Given these, the memory will be in the order of $O(N_v * k/L)$. Thus, for a fixed loading factor (default 0.57), the memory consumption of GKAGE is primarily driven by the number of variants to be genotyped. For systems with little memory, the loading factor can be increased to remove the memory consumption, but that can severely impede performance as the loading factor approaches 1. For genotyping the 28 million variants in the experiment in table 1, with an average of $3.4$ kmers per variant and a loading factor of $0.57$, the memory consumption of the index is $(28 * 10^6 * 3.4/0.57) * (8 + 4)B = 2GB$ (using 8 bytes for keys and 4 bytes for counts).

## Benchmarks

Benchmarking was performed on two different computer systems, as described in the Results section, using simulated reads for a whole genome sample with 15x coverage. A Snakemake [16] pipeline for reproducing the benchmarking results can be found at https://github.com/kage-genotyper/GKAGE-benchmarking.

# Discussion

We have presented GKAGE, a GPU accelerated, alignment-free genotyper based on a previously published CPU-based genotyper KAGE. Our results show that alignment-free genotyping is an ideal problem for GPU acceleration. While the existing KAGE genotyper is already fast by today's standards, GKAGE is considerably faster, enabling rapid genotyping even on consumer-grade computers. We see these improvements of computational efficiency as highly beneficial considering the continually decreasing cost and expanding capacity at the experimental side of whole-genome sequencing.

Since the original KAGE genotyper was implemented mainly using the array programming libraries NumPy and BioNumPy in Python, GPU support could be added to the existing code base in a clean way by using the CuPy library combined with some custom CUDA kernels with Python wrappers. We thus see our work as a strong example of how the addition of GPU support to existing tools is typically highly feasible and beneficial in cases where many independent operations are performed on an array of data, which is common for problems in computational biology. As GPUs are becoming steadily cheaper and more available, we thus see a huge potential in improving the computational efficiency of existing methods and tools, which in many cases can be achieved quite easily through the Python ecosystem with packages such as CuPy [12], Numba [17] and BioNumPy [10].

# References

1. **A New Wave of Genomics for All**
Diana Crow
Cell (2019-03) https://doi.org/gfw9g5
DOI: 10.1016/j.cell.2019.02.041 · PMID: 30901548

2. **KAGE: fast alignment-free graph-based genotyping of SNPs and short indels**
Ivar Grytten, Knut Dagestad Rand, Geir Kjetil Sandve
Genome Biology (2022-10-04) https://doi.org/grpzvf
DOI: 10.1186/s13059-022-02771-2 · PMID: 36195962 · PMCID: PMC9531401

3. **MALVA: Genotyping by Mapping-free ALlele Detection of Known VAriants**
Luca Denti, Marco Previtali, Giulia Bernardini, Alexander Schönhuth, Paola Bonizzoni
iScience (2019-08) https://doi.org/gmhw28
DOI: 10.1016/j.isci.2019.07.011 · PMID: 31352182 · PMCID: PMC6664100

4. **Pangenome-based genome inference allows efficient and accurate genotyping across a wide spectrum of variant classes**
Jana Ebler, Peter Ebert, Wayne E Clarke, Tobias Rausch, Peter A Audano, Torsten Houwaart, Yafei Mao, Jan O Korbel, Evan E Eichler, Michael C Zody, ... Tobias Marschall
Nature Genetics (2022-04) https://doi.org/grp6v6
DOI: 10.1038/s41588-022-01043-w · PMID: 35410384 · PMCID: PMC9005351

5. **Graphtyper enables population-scale genotyping using pangenome graphs**

Hannes P Eggertsson, Hakon Jonsson, Snaedis Kristmundsdottir, Eirikur Hjartarson, Birte Kehr, Gisli Masson, Florian Zink, Kristjan E Hjorleifsson, Aslaug Jonasdottir, Adalbjorg Jonasdottir, … Bjarni V Halldorsson
Nature Genetics (2017-09-25) https://doi.org/gbx7v6
DOI: 10.1038/ng.3964 · PMID: 28945251

6. **A global reference for human genetic variation**
, Adam Auton, Gonçalo R Abecasis, David M Altshuler (Co-Chair), Richard M Durbin (Co-Chair), Gonçalo R Abecasis, David R Bentley, Aravinda Chakravarti, Andrew G Clark, Peter Donnelly, … Nature (2015-09-30) https://doi.org/73d
DOI: 10.1038/nature15393 · PMID: 26432245 · PMCID: PMC4750478

7. **Scaling accurate genetic variant discovery to tens of thousands of samples**
Ryan Poplin, Valentin Ruano-Rubio, Mark A DePristo, Tim J Fennell, Mauricio O Carneiro, Geraldine A Van der Auwera, David E Kling, Laura D Gauthier, Ami Levy-Moonshine, David Roazen, … Eric Banks
Cold Spring Harbor Laboratory (2017-11-14) https://doi.org/ggmrvr
DOI: 10.1101/201178

8. **Gerbil: a fast and memory-efficient k-mer counter with GPU-support**
Marius Erbert, Steffen Rechner, Matthias Müller-Hannemann
Algorithms for Molecular Biology (2017-03-31) https://doi.org/gkzhfr
DOI: 10.1186/s13015-017-0097-9 · PMID: 28373894 · PMCID: PMC5374613

9. **GPU Acceleration of Advanced k-mer Counting for Computational Genomics**
Huiren Li, Anand Ramachandran, Deming Chen
2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP) (2018-07) https://doi.org/grp3kq
DOI: 10.1109/asap.2018.8445084

10. **BioNumPy: Fast and easy analysis of biological data with Python**
Knut Rand, Ivar Grytten, Milena Pavlovic, Chakravarthi Kanduri, Geir Kjetil Sandve
Cold Spring Harbor Laboratory (2022-12-22) https://doi.org/grp3k6
DOI: 10.1101/2022.12.21.521373

11. **Array programming with NumPy**
Charles R Harris, KJarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, … Travis E Oliphant
Nature (2020-09-16) https://doi.org/ghbzf2
DOI: 10.1038/s41586-020-2649-2 · PMID: 32939066 · PMCID: PMC7759461

12. **CuPy: NumPy/SciPy-compatible Array Library for GPU-accelerated Computing with Python**
Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, Crissman Loomis
Thirty-first Annual Conference on Neural Information Processing Systems (NeurIPS)

13. **CUDA C++ Programming Guide**
NVIDIA Corporation & Affiliates
(2023) https://docs.nvidia.com/cuda/cuda-c-programming-guide/#

14. **MurmurHash3**
Austin Appleby
(2023) https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp

15. **pybind11 -- Seamless operability between C++11 and Python**
Wenzel Jakob
GitHub (2023) https://github.com/pybind/pybind11

16. **Sustainable data analysis with Snakemake**
Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B Hall, Christopher H Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O Twardziok, Alexander Kanitz, … Johannes Köster
F1000Research (2021-04-19) https://doi.org/gj76rq
DOI: 10.12688/f1000research.29032.2 · PMID: 34035898 · PMCID: PMC8114187

17. **Numba**
Siu Kwan Lam, Antoine Pitrou, Stanley Seibert
Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (2015-11-15) https://doi.org/gf3nks
DOI: 10.1145/2833157.2833162