Implementation

Here we describe more in detail how GKAGE has been implemented. While GKAGE is implemented as part of KAGE, and in large parts uses the same code, 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 [1], a Python library built on top of NumPy [2]. 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 [3], a GPU accelerated computing library with an interface that closely follows that of NumPy. This was implemented by replacing the NumPy module in BioNumPy's modules with CuPy, effectively replacing all NumPy function calls with calls to CuPy's functions providing the same functionality, although GPU accelerated. This strategy worked out of the box for most parts of the BioNumPy solution, with only a few 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, 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 for the keys. To use the hashtable class in Python, C++ bindings are implemented using pybind11 [4]. 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 a human sample, a GPU with 4GB of memory is sufficient. More details about the implementation of the hashtable can be found in the implementation details section.

Implementation Details

GPU acceleration of existing NumPy solutions in BioNumPy

The BioNumPy Python package is implemented as a set of modules that rely heavily on NumPy for fast processing. Most of the NumPy functionality used in BioNumPy for compute-heavy tasks are based on array-operations performed on large arrays. These tasks are natural candidates for GPU acceleration, and it therefore makes sense to use CuPy to perform these tasks rather than NumPy, if a GPU is available.

Since CuPy's interface closely follows that of NumPy, it can in most cases work as a drop-in replacement for NumPy. This means that code that uses NumPy can seamlessly be modified to use CuPy simply by replacing the NumPy module with CuPy. Following is an example of how the NumPy module can be replaced with CuPy, and all calls to NumPy functions following the exchange will instead be made to CuPy's equivalent (but GPU accelerated) functions.

```
1 import numpy as np
2 import cupy as cp
3
4 # Use CuPy instead of NumPy
5 np = cp
6
7 def solution(array):
8   # This call will use CuPy's sum rather than NumPy since the modules
      are swapped
9   return np.sum(array)
```

In the code snippet above, the array sum will be computed using CuPy's sum function, which is GPU accelerated, instead of NumPy's. This is because we initially import the NumPy module as np on line 1, but then change the np variables assignment to reference the CuPy module on line 5.

Static GPU accelerated hashtable

The GPU accelerated static hashtable used to for kmer counting in GKAGE was implemented using the CUDA framework. The hashtable uses open addressing, and the data structure is a set of two arrays: one unsigned 64-bit integer array for the hashmap's keys (which are kmers), and one

unsigned 32-bit integer array for the hashmap's values (which are the kmer's associated counts).

To insert, update or query a key (kmer) in the hashtable, a simple linear probing scheme with a murmur hash is used. To find a kmer *k*'s position in the hashtable, the initial probe position p_0 is found by computing

$$p_0 = hash(k) \bmod c \tag{1}$$

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

$$p_i = p_{i-1} + 1 \bmod c \tag{2}$$

The probing will continue until either *k* or an empty slot in the hashtable is observed.

The hashtable supports three main operations: insetion, counting and querying. In each of the cases, the input is an array of 2-bit encoded kmers, and for 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.

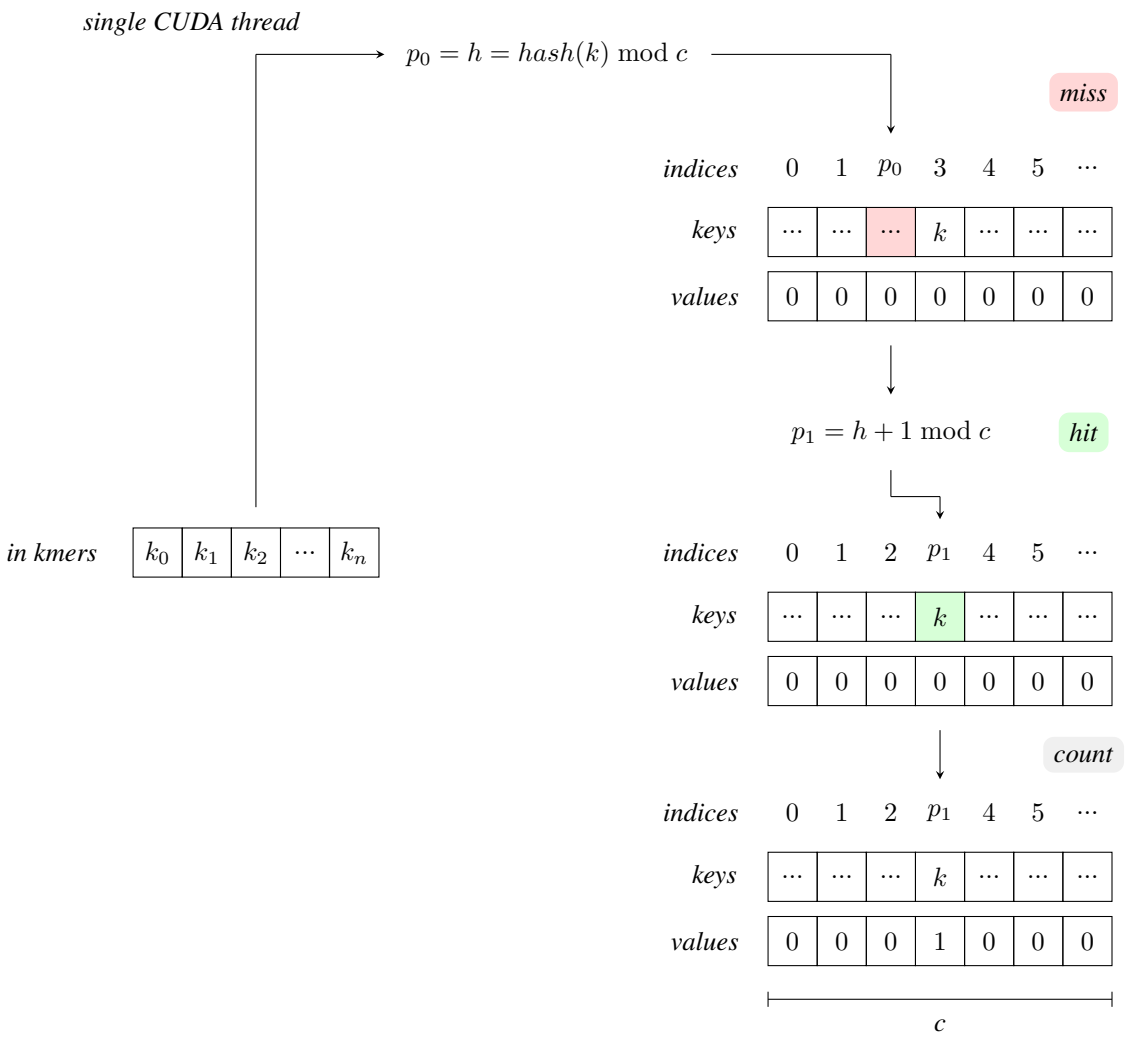

Figure 1: As an array of 64-bit integer encoded kmers are counted by the hash table, each CUDA thread will compute the first probe position p_0 for each individual kmer, and then continue probing by linearly moving up to the next consecutive slot until either an empty slot or the original kmer handled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