Fast GPU Accelerated Genotyping in Python

Abstract

Genotyping is a prominent application of high-throughput sequencing data. Recent graph-based and alignment-free genotyping methods allow for much of the compute performed in the genotyping process to be array-based. Some state-of-the-art graph and alignment-free based genotypers use array-libraries such as NumPy [1] to create performant solutions in Python. However, few options are available allowing for fast GPU accelerated genotyping in Python. Here I present a set of Python packages providing GPU accelerated functionality for graph and alignment-free genotyping, achieving up to one and two orders of magnitude speedup compared to corresponding NumPy solutions.

Introduction

Genotyping is a prominent application of the massive amounts of data provided by high-throughput sequencing. Recent developments in graph and alignment-free methods of genotyping have resulted in genotyping methods yielding competitive accuracies with significantly reduced time costs compared with their alignment-based counterparts. Despite these strides, the cost of genotyping in terms of computing power is still high. As a result, many genotyping tools are implemented using verbose, low-level programming languages such as C and C++ where the programmers need to have in-depth understanding of both hardware and low-level programming tools to create competitive solutions. However, some recent graph and alignment-free genotypers perform most of their computation in the form of array-operations. This has led to a more desireable approach, namely using array-libraries such as NumPy [1] to create performant solutions in Python where the programmer can enjoy the programmatic simplicity of Python as well as the speed of carefully optimized C code. While some options such as CuPy [2] exist to perform NumPy-like array-operations on the GPU (Graphical Processing Unit) in Python, they sometimes fall short of the desired speed increase required for a worthwhile investment in a GPU because the Python interface fails to sufficiently optimize certain operations such as nested array-expressions. Here I present two Python packages that try to circumvent this problem by providing implementations of specific functionalities used in genotyping, rather than general purpose solutions. 1) **CuCounter**: a NumPy and CuPy compatible kmer frequency counter object implemented in CUDA, allowing

for fast kmer frequency counting given a set of predefined kmers. 2) **CuStats**: a NumPy and CuPy compatible statistics module with array-expressions commonly used in genotyping.

Results

The following results were found by running experiments on a single desktop computer with an 11th Gen Intel Core i5-11400F 2.60GHz CPU and a Nvidia GeForce GTX 1660 SUPER GPU.

When benchmarking the CuCounter's counter object, we are primarily concerned with the throughput of kmers being counted rather than initialization and lookup. Several test runs using a predefined set of 153,627,144 kmers encoded as 64-bit unsigned integers, and encoded kmer data from a synthetic fasta file containing 20 million reads of length 150, yielded a throughput of roughly 1 billion kmers per second. The throughput is primarily controlled by the underlying hashtable's load factor. The counter also supports taking the reverse complement of each kmer, incrementing the frequency count for these kmers as well. Enabling the support for reverse complement counting will roughly halve the throughput.

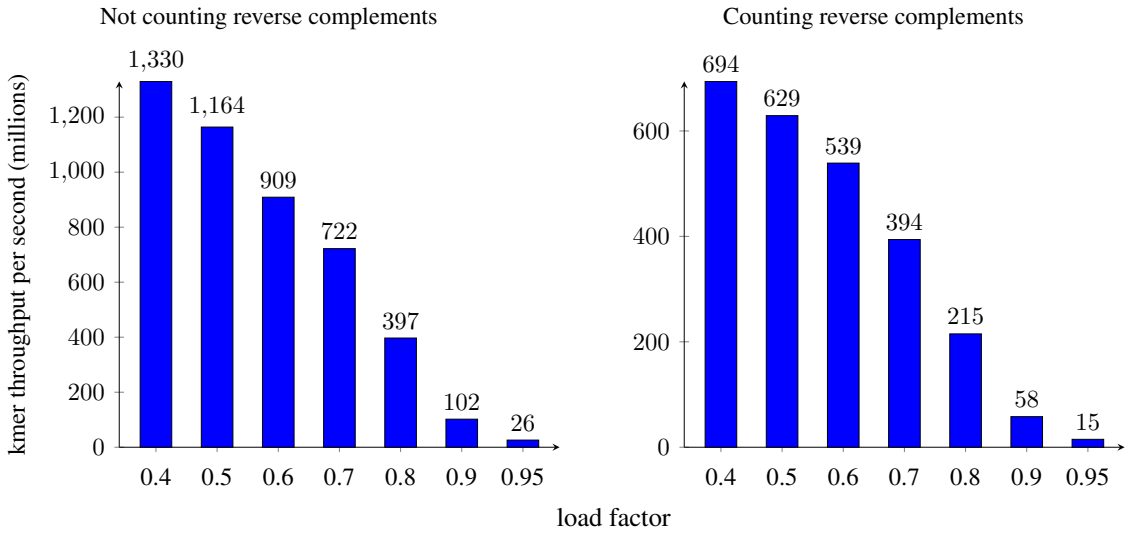

Figure 1: CuCounter kmer throughput per second (in millions) given the underlying hashtable's load factor. Left shows kmer throughput when not counting each kmer's reverse complement, while the right shows the throughput when counting each kmer's reverse complement.

CuStats will contain statistical functions useful for graph and alignment-free genotyping. Currently only two variants of a single function is are supported. The currently supported function is logpmf (log of the probability mass function). Both variants of the function have been bench-

marked against equivalent NumPy implementations for various input array sizes to demonstrate the dramatic speed increase of performing these array-operations on the GPU. For sufficiently large arrays, CuStats achieves up to 120X speed increase over NumPy.

Implementation details

Both CuCounter and CuStats are designed to be compatible with both NumPy and CuPy. Both NumPy and CuPy are in fact prerequisites for CuCounter and CuStats, as the supported features in both packages currently only interact with NumPy and CuPy arrays. All current features in both packages will accept both NumPy and CuPy arrays and will always return arrays of the same type as the input. When NumPy arrays are used, both packages will handle copying of the underlying array data to and from the GPU. Naturally, using CuPy arrays will result in significantly better performance, as their underlying data will already reside in GPU memory.

CuCounter's underlaying hashtable

The underlaying hashtable used in the CuCounter's counter object follows a simple open addressing with linear probing scheme. When an array of 64-bit encoded kmers are provided, each kmer is inserted by a single block thread in a kernel call. The thread will deploy a simple linear probing scheme and search the hashtable for either an empty position, or a position occupied by the provided kmer. In this search, the initial position p_i for kmer k is found by computing

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

and every consecutive position p_{c+1} following position p_c is determined by computing

$$p_{c+1} = (p_c + 1) \bmod c \tag{2}$$

where $hash$ is a murmur hash function and c is the hashtable's capacity. The search terminates when either k or an empty slot is observed in the hashtable. For both insertion (during initialization) and counting, CUDA atomic operators are used to ensure validity.

CuStats functionality

The CuStats package is meant to be a collection of useful statistical functions used in (graph and alignment-free based) genotyping. While these functions can be effortlessly implemented using NumPy, and thereby CuPy to leverage the GPU, their performance suffer because of the general-purpose nature of NumPy and CuPy. For instance, when creating nested array-expressions, CuPy will often perform many temporary array allocations, and needlessly move data back and fourth between registers and global memory. Both of these problems are circumvented in CuStats by implementing targeted kernels where the entirity of the expressions can be efficiently computed.

Discussion

I have here presented two new Python packages, CuCounter and CuStats, providing GPU support for typical computational tasks performed in genotyping. Both packages are works in process, and changes are overwhelmingly likely to be made to both.

Interesting future work may reside in optimizing the underlying CUDA hashtable of the CuCounter counter object, possibly leveraging previous work on static hashtables using buckets, such as in BGHT [3], in order to allow for high performance even with high load factors. While this table is a useful tool for anyone who wants to count kmer frequencies with GPU support in Python, it is not meant to compete with potentially faster CUDA hashtables existing elsewhere, even for the particular use of kmer counting.

References

[1] 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. URL: https://doi.org/10.1038/s41586-020-2649-2.

[2] 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. URL: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

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