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. We have taken a basis in the current state-of-the-art graph-based and alignment-free genotyper KAGE [2] and developed a GPU supported version called GKAGE. I also show that GKAGE, running on a GPU, achieves up to ... speed up over KAGE, running on a CPU.

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. One such genotyping tool is the current state-of-the-art graph-based and alignment-free genotyper KAGE [2], which is written in Python and heavily leverages NumPy. We have taken KAGE as a basis and developed a GPU supported version, GKAGE, showing that genotyping tools performing most of their compute as array-operations can benefit greatly from GPU support. This was achieved by examining KAGE to find bottlenecks in its compute pipeline, and then creating GPU supported drop-in alternatives.

Implementing GPU support for KAGE

In order to prioritize which components in KAGE to implement GPU support for, the genotyping pipeline was examined in order to find bottlenecks that seemed appropriate to outsource to the GPU. For example, if a component performed many large array computations, it would be deemed appropriate for outsourcing to the GPU as it would be fitting of the type of compute work that GPUs are especially designed to be superior for. Three components were identified and have had GPU supported alternatives implemented.

Reading and encoding kmers from fasta files is achieved in KAGE by using BioNumPy [3], a Python library built on top of npstructures [4] and NumPy [1]. 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. In order to add GPU support to this step, I utilized CuPy [5], a GPU accelerated alternative to NumPy, as a drop-in replacement for NumPy. This was achieved by adding a function to the BioNumPy module that replaces all relevant NumPy modules in BioNumPy's sub-modules, with CuPy. This yielded GPU support (and significant speedup) to the reading and encoding of kmers in BioNumPy, all while preserving the BioNumPy code and without having to implement complicated GPU kernels ourselves.

Counting kmer frequencies observed in the encoded kmer chunks from the previous step has received GPU support through a Python counter object that is built on top of a parallel hash table implemented in CUDA. The Python counter object provides an API consisting of the following: 1) *initialization* from either a NumPy or a CuPy array of 64-bit encoded kmers, 2) *counting* of kmer frequencies observed in either a NumPy or CuPy array of 64-bit encoded kmers where observed kmers not present in the initialization array are ignored, 3) *lookup* of observed kmer frequencies provided a NumPy or CuPy array of 64-bit encoded kmers to look up. Whilst the entire API supports both NumPy and CuPy arrays, CuPy is preferred as it circumvents the need to copy data to and from the GPU memory.

Computing the log of the probability mass function for many large arrays was perhaps one of the most GPU suited tasks in KAGE. Since KAGE's native implementation uses NumPy, one option would be to use CuPy as a drop-in replacement for these array-operations. However,

these array-operations happened through several nested steps, resulting in several unnecessary and significant memory allocations both when using NumPy and CuPy. Therefore, I implemented a targeted kernel performing this compute efficiently in CUDA.

Results

Together, the three implementations described above have been incorporated into KAGE, resulting in GKAGE (GPU KAGE), yielding up to ... speedup over KAGE running on a ... CPU with ... cores, when running on a ... GPU.

This work has further resulted in partial GPU support of BioNumPy, as well as two Python packages: *CuCounter*, found at `https://github.com/jorgenwh/cucounter`, providing an easy to use static GPU hash table, and *CuStats*, found at `https://github.com/jorgenwh/custats`, providing the log of the probabiliy mass function with GPU support in Python.

Discussion

While adding GPU support to KAGE have resulted in the potential for significant speedup for users with a Nvidia GPU, the current implementations come with a couple of caveats. Both the CuCounter and CuStats packages are only supported by Nvidia GPUs. Furthermore, the goal of this work has not been to improve upon or compete with state-of-the-art implementations for the respective processes implemented here, but rather to provide easy-to-use Python implementations that could be used in KAGE to speed up state-of-the-art graph-based and alignment-free genotyping in Python for users having access to a Nvidia GPU.

Further work may naturally lie in implementing support for GPUs created by other manufacturers, as well as exploring more sophisticated static GPU hash table schemes.

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] 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. URL: https://www.biorxiv.org/content/early/2021/12/20/2021.12.03.471074.

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

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

[5] 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.

Implementation details

CuCounter's underlying hash table

The underlying hash table used in CuCounter's counter object follows a simple open addressing with linear probing scheme. When an array of 64-bit integer encoded kmers are provided, each kmer is handled by a single CUDA thread in a kernel call. The threads, all employed to either insert (during initialization) or search for (during counting and lookup) a single kmer, will deploy a simple linear probing scheme and search the hash table 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.