Parallelizing DNA Sequence Read Alignment in Python

Trevor Ridgley

Department of Biomolecular Engineering & Bioinformatics

University of California, Santa Cruz

1. Background

Traditionally, genetics researchers analyzed inheritance patterns in plants and animals by performing breeding experiments. Later approaches in molecular biology deepened our understanding of inheritance by connecting protein interactions to observations between treatment and control groups in different species. More recently, the field of genomics investigates hereditary mechanisms using information encoded directly in DNA. The primary source of this data is genetic sequencing, and next generation DNA sequencing is a highly parallel process that can generate on the order of hundreds-of-millions of DNA polymer reads. Each read typically consists of only a few hundred bases A, C, G, or T while cellular genomes typically span millions or billions of bases. Consequently, assembling all of these reads into a complete genome de novo or mapping them back to a reference genome by finding common substrings is computationally intensive. Therefore, read mapping is a hot field for development of powerful new algorithms like MiniMap2 [1] that builds upon the seed and extend algorithm from BLAST [2].

2. Introduction

A large number of Bioinformatics applications are implemented using Python for its approachable syntax, standard data manipulation capabilities, and powerful libraries like numpy, pandas, and scipy. There are also several machine learning packages including Scikit Learn, PyTorch, and Tensorflow. For these reasons, I wanted to explore CUDA packages that are available for Python and evaluate their ease-of-use and performance in the context of a sequence alignment algorithm. The two options under consideration were Numba [3] and PyCUDA [4], but I became concerned about investing my time learning PyCUDA when they recently announced that platform support was discontinued. I want to re-implement a DNA sequence read local alignment program called simpleMap.py, particularly the seed finding and sequence scoring code, using GPU kernels. My plan is to compare the runtime performance between 4 configurations: Sequential read alignment, sequential read alignment pre-compiled with Numba, parallelized kernel read alignment with Numba, and parallelized GPU kernel plus CPU multithreading.

3. Methods

2.1 Approach

The initial step of a seed and extend algorithm (like MiniMap2) is creating a collection of seeds. In sequential implementations, a sliding window of size W iterates over all possible $M - W + 1$ offsets that window w_i can assume, where M is the length of each read being aligned. Within each window w_i, we collect all of the k-mers k_{ij} of size $K,\ 0 < K \leq W,$ and sort these lexicographically. The smallest lexicographic k-mer k_{im} can

be chosen as a seed that we call the ‘min-mer’. This process is repeated for every read that we are trying to map (which can be millions depending on the sequencing coverage or depth), and again for the reference genome which typically thousands-to-millions of times larger than each of the reads. The seed generation process is embarrassingly parallel, so we can process every window using a GPU thread to speed up this phase of the algorithm by a factor of $M - W + 1$ windows for each read that we are mapping. The efficiency gained by generating seeds from the reference genome is theoretically $(L - W + 1) / T$, where L is the reference genome length in bases and T is the # of threads available for simultaneous computation.

The next phase extends the seeds into connected components of a sequence graph for each of the reads G_{i-read} and its analog in the reference sequence G_{j-ref}. Then, these graphs are evaluated using a scoring matrix of dimension $g_i \, x \, g_j$ to determine the best position that our read maps to in the reference genome using the Smith-Waterman local alignment algorithm. Although Smith-Waterman is a dynamic programming algorithm that cannot be parallelized trivially, this computation must be repeated for every read from our seed extension phase. Scores must be calculated for each of the alignments and the reference genome position with the best score is used for mapping that read. So, for each read in our sequencing data collection, we can launch a GPU thread to perform these scoring calculations that are $O(g_i g_j)$.

2.2 Dataset

A FASTA file containing the first 9 million base pairs (9Mb) of human chromosome 3 can be downloaded from the hg19 human genome assembly at the UCSC Genome Browser here: https://genome.ucsc.edu/s/tridgley/hg38_chr3_9mb
An illumina sequencing run with 500 reads that map to this region are provided in the file NA12878.ihs.chr3.100kb.1.fastq.tiny (see acknowledgements section) which is a subset of the NA12878 sample from the 1000 Genomes Project [5][6].

2.2 Libraries and Tools

Various C language bindings exist for Python, such as Cython. One option was to implement GPU kernels directly in C and call them through these bindings, but I preferred to approach this project as a computational biologist who wants to parallelize computation without knowledge of C programming. I found several libraries that enable GPU parallelization using Python, including Numba [3] and PyCUDA [4].

2.3 Environment Setup

All computation for this project was performed on the Lux compute cluster at UC Santa Cruz. Lux is a high-performance environment featuring 108 nodes with 2 20-core Intel Xeon Gold CPU's and 2 NVIDIA Volta 100 GPU's per node. As our class storage space was limited, the full 5.4Gb Anaconda toolkit installation was too large, so miniconda3 at just 3.4Gb was chosen instead. Note, an older version of cudatoolkit 9.0 was required to run with our Lux environment 9.4 drivers; the latest cudatoolkit 9.5 was incompatible.

wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
chmod u+x Miniconda3-latest-Linux-x86_64.sh
./Miniconda3-latest-Linux-x86_64.sh
Add path to miniconda3/bin
conda install Numba
conda install -c bioconda pysam
conda install -c anaconda cudatoolkit=9.0
conda install -c bioconda pysam
module load cuda10.0
pip install pycuda
module load slurm

2.4 Pre-Compiled Numba Decorators

See code at https://github.com/tridgley/ParallelReadMapper
Run command: python3 simpleMapJitClass.py hg19.chr3.9mb.fa NA12878.ihs.chr3.100kb.1.fastq.tiny --t 30 > result.txt

The Numba library provides different levels of optimization that vary in terms of performance and ease-of-use. I wanted to compare the performance gain between a relatively simple just-in-time (jit) decorator-based approach that pre-compiles the python code versus a more traditional kernel function implemented using the jit.cuda framework. According to the Numba documentation, numeric Python functions that are preceded by the decorator @jit can be automatically pre-compiled and parallelized [7]. The library automatically takes advantage of pre-compiled speed, multi-threading and vectorization on the CPU. This works best with Python code that uses Numpy, but our aligner mostly uses standard python data types like dicts and lists to process String comparisons. Another complication is that object-oriented classes are used by our aligner quite heavily, so the auto-jit and njit (jit(nopython=True) decorators are insufficient.

To circumvent these issues, the experimental decorator @jitclass was used for creation of the minimizer index for both reference genome and sequence reads to be aligned. The documentation for this API states that Python classes can be preceded by the @jitclass(spec) decorator, where spec describes the class attributes using a c-context to aid the compiler [8]. The MinimizerIndex class contains some native types like unicode for the reference and read DNA sequences, and int for each of the window width W, k-mer width K, k-mer max count of T. But the results of min-mer search are stored in

dictionaries of lists that require more complex specifications for the pre-compiler to work properly.

Numba provides another API for container types like Python list and dictionary called typed, so these were used for the minimizerMap and minmerOccurrences dictionaries [9]. The container declaration signatures appear below along the class spec:

```
mm_kv_types = (types.unicode_type, typed.List(types.int64))
mo_kv_types = (types.unicode_type, types.int64)
miSpec = [
    ('targetString', types.unicode_type),
    ('w', types.int64),
    ('k', types.int64),
    ('t', types.int64),
    ('minimizerMap', types.DictType(*mm_kv_types)),
    ('minmerOccurrences', types.DictType(*mm_kv_types)) # *mo_kv_types))
]
```

The last step was replacing each of the Python lists with calls to the Numba equivalent of dtype.List() for the code to successfully compile and run without errors. These changes were made in each of the MinimizerIndexer class methods: __init__() and getMatches().

3.5 Numba CUDA Kernels

See code at https://github.com/tridgley/ParallelReadMapper
Run command: python3 simpleMapKernel.py hg19.chr3.9mb.fa NA12878.ihs.chr3.100kb.1.fastq.tiny --t 30 **--g** > result.txt

Numba provides an API for writing CUDA kernels in Python and that allow parallel execution on GPU [10]. The decorator @cuda.jit precedes a kernel function, but when I tried to implement the kernel function as a method of the MinimizerIndexer class, there was an error stating that this feature is unsupported [11]. After trying several different work-arounds, including static member and alias member functions that call a global kernel, I was forced to revert back the jitclass changes and implement cuda.jit solely as a global device function.

Fortunately, the MinimizerIndexer class methods __init()__ and getMatches() were updated from sequential algorithms to employ the same device kernel for computing both the reference genome and individual sequence read min-mer collections. The kernel configuration was set to 1024 threads per block and $int(M/1024) + 1$ for sequence reads and $int(L/1024) + 1$ for the reference genome sequence. The sequence read and reference sequences were copied to the GPU device global memory using d_sequence = cuda.to_device(sequence), but this resulted in errors saying that character strings are unsupported. It looks like cuda.jit only supports numeric types, so it was necessary to convert the DNA sequences to their ASCII character codes and pass the array of byte

values to the kernel. A numpy result array of length $L - W + 1$ was passed to the kernel to capture the indices for which min-mers were found in each window w_i.

The CUDA kernel findMinmersInWindow() takes arguments for a device pointer to the byte array representing the DNA sequence, int w for the window size, int k for the k-mer size, and a device pointer to the array of integers that represent the minimal lexicographic k-mer in each window. The Numba cuda API provides direct links to the blockIdx, blockDim, and threadId attributes for calculating the flattened thread ID. Each thread performs a sequential operation over the window w_i by checking whether each k-kmer k_j within the window is minimal. Each time that the minimal k-mer is found, the first element of the k-mer k is stored to the arable minPos, and after all possible k-mers of that window are evaluated, the lowest is written to the global result array at the position of the thread id.

3.6 Numba CUDA Kernel & CPU Multi-threading

See code at https://github.com/tridgley/ParallelReadMapper
Run command: python3 simpleMapKernel.py hg19.chr3.9mb.fa NA12878.ihs.chr3.100kb.1.fastq.tiny --t 30 > result.txt

The DNA sequence read mapper call structure relies on the creation of several objects via the simpleMap() function call that would make a kernel implementation quite difficult. Additionally, distributing these M reads among M different kernels would be inconsistent with a divide-and-conquer approach because each read represents a different sequence to be mapped and scored. Because of these challenges, it was determined that CPU multi-threading via the threading or multiprocessing Python packages would be best given this architecture.

Using this implementation, there was no observed performance increase, presumably due to the nature of the Python Global Interpreter Lock for CPU-intensive computations [12]. Even after converting the simpleMap function call to a class that might avoid call collisions between threads, there was still no runtime reduction. Instead, pools from the Python multiprocessing package were used to distribute simpleMap function calls among different processes. Again, this resulted in poor performance due to the overhead of generating enough pools for the number of reads, even with up to 40 CPU's available on the Lux GPU node. Finally, the Process and Queue packages from the Python multiprocessing library were used to divide the reads into parallel CPU threads and this resulted in a substantial performance gain compared to sequential.

4. Results

4.1 Sequential Runtime

The standard application performs the following operations:

- Reads the FASTA files containing human genome reference sequence and NGS reads using PySam.
- Creates the minimizer index for both reference sequence and NGS reads.
- Creates a graph of connected components for the sequence reads that map to the reference genome.
- Calls the Smith-Waterman local alignment scoring algorithm for each connected component of the alignment graph.
- Prints the best alignment for each read to the console.

The runtimes for each step of this workflow were logged using time.time() Python package, and results before any enhancements are summarized in the Table 1:

Table 1: Sequential operation runtimes for simpleMap

Operation	Runtime (sec)
Compute min-mer index from reference genome	39.63
Compute min-mers from reads and create seeds	(500)(1.97e-4)
Find connected components of seed graph	(500)(2.33e-4)
Smith-Waterman	(500)(0.13)
Total	114.86

We can see that the biggest contributor to runtime is the reference genome seed index computation that requires about 40 seconds or 34.5% of the total 115 second runtime. This represents the greatest opportunity for parallelism via GPU because every window of the 9Mb genome could be searched for its lowest k-mer in an embarrassingly parallel approach. The next biggest contributor to runtime is the sequential execution of every read against the reference genome. The component steps cannot be parallelized because the Smith-Waterman scoring depends on the graph structure, which depends in turn on the min-mer seeds from both the read and the reference genome (which was already computed once). The total runtime of 115 seconds was the best that was observed after some optimizations to the min-mer index finding. Initially, simpleMap

required approx. 180 seconds to complete. The current implementation requires approx. 135 seconds runtime on my personal 2019 Macbook Pro, but Lux is considerably faster

4.2 Pre-Compiled Runtime using Numba jitclass

I expected the pre-compiled code to run marginally faster than the sequentially interpreted Python code, but surprisingly it was about the same. There are two modes for jit pre-compilation, object mode and nopython mode. I used the nopython (njit) mode that should employ vectorization over any numpy arrays or loops to optimize sequential operation, in addition to the intrinsic speed of a compiled versus interpreted runtime. A significant change within this class was the requirement for Numba List and Dict structures that are prominent in implementation.

Table 2: Numba jit pre-compiled operation runtimes for simpleMap

Operation	Runtime (sec)
Compute min-mer index from reference genome	69.82
Compute min-mers from reads and create seeds	(500)(0.001)
Find connected components of seed graph	(500)(0.003)
Smith-Waterman	(500)(0.13)
Total	142.31

We can see that computation of the reference sequence min-mer index took substantially longer at 69.82 seconds, or 71% longer than the sequential version. I suspect that the efficiency of the Numba Dict containing List types suffers from considerable overhead versus the Python variants because they are built upon completely different data structures. Also, these are some of the more efficient structures of base Python. Jitclass is guaranteed to compile in nopython mode instead of object node, so there is no chance that the interpreter is running Python for these code segments in the background. Another huge drawback of the Numba dictionary is the lack of dict.get() to test for keys, without which we could crash. Instead, I had to use key in dict which seems to iterate much more slowly than Python's dict.get().

4.3 GPU-Enabled Runtime using Numba cuda.jit

Overall, the mapping of 500 reads saw a runtime improvement of 24 seconds or 21%. A runtime gain similar to that for the pre-compiled njit decorator was observed for both the creation of sequence read seeds and the connected seed graph. If we zoom in on the loop that calculates the minimizer seeds themselves, the improvement was about 100x.

Table 3: Numba jit.cuda GPU operation runtimes for simpleMap

Operation	Runtime (sec)
Compute min-mer index from reference genome	11.79
Compute min-mers from reads and create seeds	(500)(2.60e-5)
Find connected components of seed graph	(500)(9.44e-5)
Smith-Waterman	(500)(0.13)
Total	90.62

We can see that the overall reference genome min-mer index creation required 11.79 seconds instead of 39.63 seconds for the fully sequential algorithm, an approx. 4x runtime improvement. But due to limitations of cuda-jit, particularly its lack of support for string and dict types in cuda kernels [13], mapping the compatible ASCII int array back to character strings that could be used as the MinimizerIndexer keys necessitated sequential computation that stifled an impressive potential gain.

4.4 GPU-Enabled Runtime with Numba cuda.jit and CPU multiprocessing

At this point, I was somewhat frustrated by the minimal runtime improvement of my Python sequence read mapper, and the most glaring issue remaining was the sequential alignment of all 500 reads. It would not have been possible for me to write a kernel that handles all of the class objects and their Python containers with my current understanding of the library and its limitations. In fact, if I was going to that length then I would just implement it using C. Instead, I chose to implement CPU multi-threading for the read alignments because it maintained the code structure and seemed logical given that each read uses different sequence data. These are the results:

Table 3: Numba jit.cuda GPU operation runtimes for simpleMap

Operation	Runtime (sec)
Compute min-mer index from reference genome	11.79
Compute min-mers from reads and create seeds	(500/10)(1e-4)
Find connected components of seed graph	(500/10)(1e-5)
Smith-Waterman	(500/10)(0.13)
Total	30.62

I ran the batch job with 10 CPU's (out of the 40 that are available on our Lux compute node) and saw a roughly 3x runtime improvement for the read alignments alone. Of

course, a 10x improvement for this operation would have been ideal, but of course there are more limitations that I encountered for this approach as well. The most glaring was a complete lack of support for Numba GPU acceleration when called from parallel Python threads or processes [14]. Something about the way that the processes are spawned via forking prevents them from communicating effectively with cuda. Hence, any attempt at this resulted in a CUDA_ERROR_NOT_INITIALIZED [15]. Unfortunately, I was forced to revert the code for getting seeds back to sequential computation for each read instead of using the same cuda-jit kernel as the reference genome sequence, but even with this trade-off the gain was worth it.

As a side note, when I tried both threading and pool implementations of CPU parallelism, they were slower than the 90 second runtime for cuda.jit alone. Threading required 100 seconds to map 500 reads but at least seemed compatible with the existing cuda.jit kernel. Pools were the worst CPU threading option for this application requiring 123 seconds to map just 10 reads and incompatible with the cuda.jit kernel during the sequence read min-mer phase. Pools often timed out and crashed as well, so these approaches were thrown out in exchange for the processes.

5. Discussion

Further parallelization of the reference genome min-mer indexer could have been achieved by maintaining the ASCII code format instead of converting min-mers back to DNA sequence sub-strings, but a solution would still be necessary for the min-mer index dictionary structure that is also unsupported by cuda.jit. Even if an alternate structure was decided, this would be counterproductive because the strings are needed later for building the connected graph from seeds for each of the reads, so these tasks would need conversion to ASCII codes as well. But the gains to be had by implementing these work-arounds are relatively small, possibly 10-12 seconds, which seems pointless when the functions could have been implemented in C where hashmap and string types are supported, preventing ripple-effects throughout the code. The bigger (and more warranted) challenge would be parallelizing the read alignments and this is still an outstanding question that I have, even in C. I considered concatenating a single large array with all of the sequence read connected graphs and writing this array along with the connected reference graphs to a GPU device global memory. The challenge here would be knowing where each graph begins and ends since they are different sizes, but this could be addressed with a 3rd binary array similar to the generalized sum/scan approach. The Smith-Waterman scoring could also be done this way with the base character comparisons being done in ASCII rather than string formats. I expect these changes would have resulted in total runtimes in the single-digit seconds range.

6. Conclusion

Although Python is a very attractive programming language for many reasons, it certainly has its limitations. Employing a combination of Numba GPU and Python multiprocessing packages yielded only a 75% runtime improvement compared with a strictly sequential computation. I feel that this muted achievement is mainly due to my inexperience working with all of these packages, navigating the limitations of each package, and trying to apply these approaches to an application of non-trivial complexity. Most of the example code for these libraries deals with very specific calculations that are time-consuming but relatively simple in terms of function and code structure. On the other hand, the read mapper leverages many of Python's different container types in an object-oriented context which made parallelization from the top-level very difficult for a beginner. Although I learned a lot from this project, my biggest takeaways are to implement intensive algorithms like DNA sequence read mapping with a programming language like C and not to waste too much time making a scripting language like Python do work that it was not intended or optimized for. Python does great in a data storage and manipulation context, but is less than ideal for massive parallelization schemes.

Acknowledgements

The code scaffold was originally written by Dr. Benedict Paten, Associate Professor of Biomolecular Engineering & Bioinformatics at UC Santa Cruz, and I adapted it from Python 2 to Python 3. The reference genome FASTA file hg19.chr3.9mb.fa and 500 sequence reads contained in NA12878.ihs.chr3.100kb.1.fastq.tiny were also provided by Dr. Paten. The sequential algorithms for MinimizerIndexer, ClusterSeeds, and Smith-Waterman were implemented based on course material from BME-230A at UC Santa Cruz and the MiniMap2 paper [1]. Installation instructions for miniconda were provided by Cade Mirchandani of the Corbett-Detig Lab at UC Santa Cruz. I would also like to thank Dr. Steven Reeves for his feedback on this project, particularly the Numpy C bindings for Python and our email discussion about parallelizing the Smith-Waterman scoring functions on GPU vs CPU threading.

Supplement

Raw output for sequential aligner:

```
2020-06-10 19:28:16,066 - root - CRITICAL - Finished alignments in 115.45513129234314 total seconds, average alignment score: 188.95
```

Raw output for jitclass aligner:

```
2020-06-10 16:36:24,816 - root - INFO - Finished alignments in 142.3134264945984 total seconds, average alignment score: 188.95
```

Raw output for cudajit:

```
2020-06-10 19:02:53,546 - root - INFO - Finished alignments in 90.66405987739563 total seconds, average alignment score: 189.014
```

Raw output for cudajit & subprocess:

```
2020-06-10 16:51:38,815 - root - INFO - Finished alignments in 41.76912045478821 total seconds, average alignment score: 190.47379032258064
```

References

[1] Li, Heng, and Inanc Birol. “Minimap2: Pairwise Alignment for Nucleotide Sequences.” *Bioinformatics* 34.18 (2018): 3094–3100. Web.

[2] Altschul, Stephen F et al. “Basic Local Alignment Search Tool.” *Journal of Molecular Biology* 215.3 (1990): 403–410. Web.

[3] https://developer.nvidia.com/how-to-cuda-python

[4] https://developer.nvidia.com/pycuda

[5] https://www.internationalgenome.org/data-portal/sample/NA12878

[6] Genomes Project Consortium et al. “A Global Reference for Human Genetic Variation.” (2015): n. pag. Web.

[7] https://numba.pydata.org/numba-doc/latest/user/5minguide.html

[8] https://numba.pydata.org/numba-doc/dev/user/jitclass.html

[9] https://numba.pydata.org/numba-doc/dev/user/jitclass.html#specifying-numba-typed-containers-as-class-members

[10] http://numba.pydata.org/numba-doc/latest/cuda/kernels.html

[11] https://github.com/numba/numba/issues/4451

[12] https://realpython.com/python-gil/

[13] https://stackoverflow.com/questions/56866190/numba-kernel-is-not-allowing-dictionaries-or-string-functions

[14] https://github.com/tensorflow/tensorflow/issues/8220

[15] https://nyu-cds.github.io/python-numba/05-cuda/