master’s project notes

Contents

1 Examining counter implementations

1.1 The npstructures counter object

The counter object found in the npstructures package is a specialized hashtable used for counting frequencies of unique 64-bit data values. The original npstructures counter object and its underlying hashtable are implemented in Python, and therefore rely heavily on NumPy to perform its underlying functionalities efficiently. Describe the npstructures counter object ...

Diagrams?

1.2 CUDA counter

One of the primary issues with the npstructures counter object is that its implementation is constrained by the functionalities provided by NumPy for good performance. This results in a sequence of, perhaps unnecessary, array operations for initialization, counting and lookups. In an attempt to mitigate this, I implemented a simpler and more straightforward counter object in CUDA and used the pybind11 library to create Python bindings for the implementation as to create a drop-in replacement for the npstructures counter object.

Describe the CUDA counter implementation ...

1.3 Performance comparison

...

Comparison between different hashtable capacity sizes for the CUDA hashtable implementation. We expect faster initialization, counting and lookup when the capacity is larger. The following bar chart show the time spent processing counts in the hashtable for 20 million reads of 150 bases. The counter object was initialized with 20 million unique kmers as keys, and the fasta file containing the reads was in chunks of 10 million bytes using BioNumPy. Since the counter object was initialized with 20 million keys, the theoretical smallest possible capacity of the hashtable was 20 million slots. The tests benchmarked the total time spent waiting for the count call, summarized from all the chunks making up the 20 million reads.

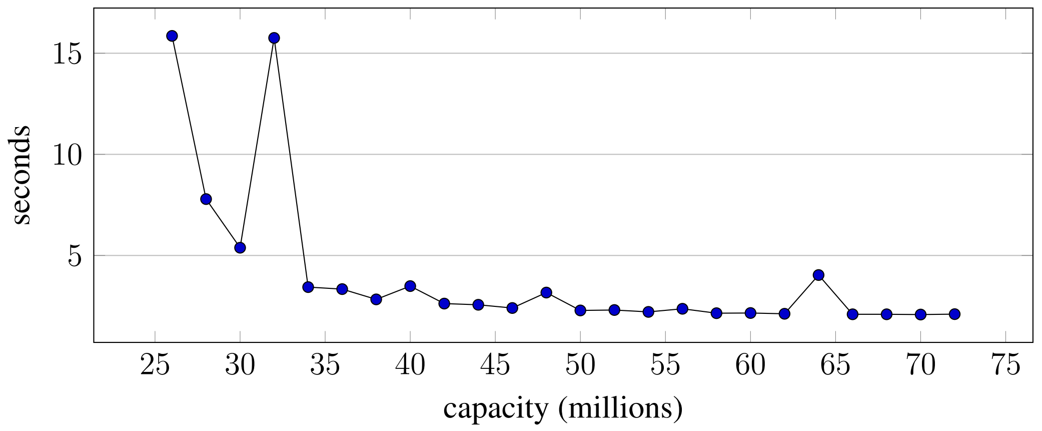

Figure 1: The total time spent waiting for the count calls when counting 20 million reads of 150 bases each. As expected, the tradeoff between memory and efficiency becomes apparent as larger capacities result in faster counting times. Although we quickly reach a point of dimishing returns, we see small improvements all the way up to a capacity of 70 million slots, with a capacity of 70 million slots spending a total of 2.079 seconds counting 20 million reads. The benchmark was only ran from 26 million up to 72 million slots. Time consumption grew exponentially with smaller capacities than 26 million for this counter of 20 million keys.

...

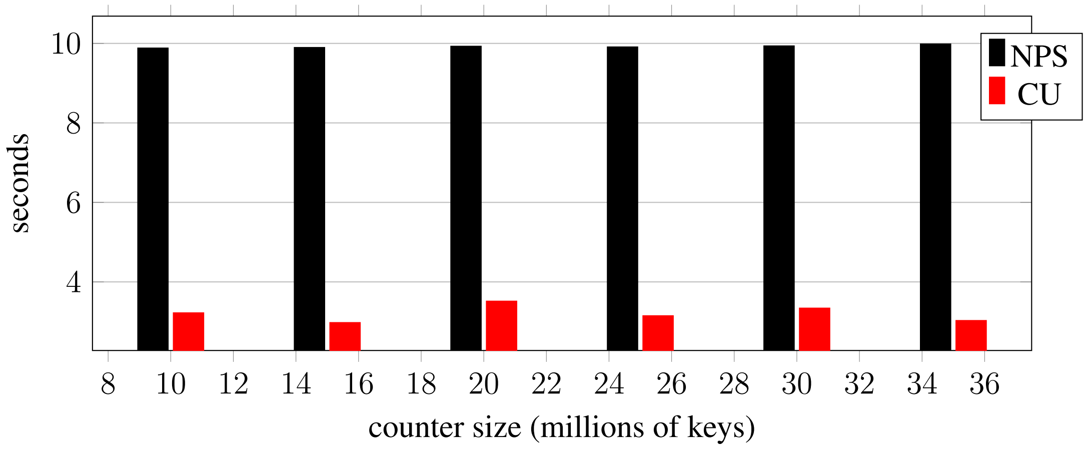

Figure 2: CUDA counter uses a capacity factor of 2.0.